CONSTRUCTION OF INTERFACES FOR THE TRANSFER OF DATA BETWEEN GEOGRAPHICAL INFORMATION SYSTEMS

A THESIS
SUBMITTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE
OF
DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE
IN THE
UNIVERSITY OF CANTERBURY
BY

RICHARD TERRENCE PASCOE

University of Canterbury
1994

Abstract

This thesis deals with the problem of constructing the interfaces needed to transfer data between different geographical information systems.

Traditionally, data values were moved from one computer system to another using magnetic tapes and, more recently, using CD-ROM. Use of communication networks to move data sets from one computer system to another greatly widens the scope for data transfers. Examples of types of data transfer now possible are: the development of federated and distributed database management systems which allow data to be transferred among databases stored on different computer systems; the use of electronic mail servers for the distribution of data files; use of hand-held data collectors that can transfer field data through communication links; and applications which, though executing on separate computer systems, communicate with each other through a network.

The diverse data representations defined for different geographical information systems cause difficulties when attempting to transfer geographical data. To better understand these representations, and thereby assist in the construction of interfaces, the representation of geographical data values is discussed using the concepts of: an abstraction, which is a collection of ideas about the data to be represented; a data model, which defines a notation for describing the types of data values, and operations for manipulating these values; and a schema, which is a definition of an abstraction using the notation and operations defined by a data model.

The author's approach to interface construction is based on generating interfaces from formal transfer specifications. Any such specification defines: the various representations to which the data conforms before, during, and after the transfer; the transformations to modify the data values from one representation to another; and perhaps also transformations to move the values between different computer systems.

An interface can be generated from such a transfer specification using the software tool `a2b`, which has been developed by the author as part of this project. An interface is defined as comprising some combination of four types of interface modules. *Decoder* modules transform

data values from a text-file representation into a memory-resident representation, and *encoder* modules perform the reverse transformation. *Translator* modules modify the types of data values, and *communication* modules send and receive data values through a communications network. These modules are either generated directly by `a2b`, or from specifications generated by `a2b` and processed by other software tools such as `bison`, `flex`, and the software tools `rosy` and `pepsy` provided within the ISO Development Environment (ISODE).

Contents

List of Tables

List of Figures

Introduction

Chapter 1

Very large databases that describe spatial and non-spatial aspects of the real world have been built up in various geographical information systems. Capturing this data for storage in a geographical information system has been a laborious and error-prone task which has accounted for perhaps 80 % of the cost of many projects (Star & Dickinson 1990). Often, large parts of the data stored in one geographical information system are required for many other geographical information systems. The definitions of coastal, national, regional, road, and property boundaries, for example, are used by utility organisations as a framework for their power, drainage and telephone networks, and by local and national government organisations for administration.

To avoid the need for different organisations to duplicate the capture of the same data, methods have been developed by which data captured by one organisation can be transferred to others. These methods are used in software packages called interfaces, which transfer data from one geographical information system to another. The objective of the project described in this thesis is to simplify the construction of these interfaces.

Transferring data from one organisation to another is difficult where, as is typically the case, the two organisations use different geographical information systems, each with its own data representation. Thus, data transfers frequently require a transformation of data from one representation to another. Even if the two organisations use the same type of geographical information system, data values may have to be changed because the organisations have an interest in different aspects of the same real-world entities. For example, one organisation may be interested in aspects of a road such as the type of surface, the date last surfaced, and the number of lanes, while the other organisation may only be interested in the road because it defines part of a census block boundary, in which case the aspects of the road previously mentioned are of no interest.

A principal reason for the difficulties in transferring geographical data is therefore the diverse representations that are used for geographical data. To gain a better understanding of these representations, and thereby assist in the construction of interfaces, geographical data representations are discussed in Chapter 2 using the concepts of: an abstraction, which is a collection of ideas about the data to be represented; a data model, which defines a notation for describing the types of data values, and operations for manipulating these data values; and a schema, which is a definition of an abstraction using the notation and operations defined by a data model.

Another factor complicating data transfers is the need to move data from one computer system to another. In the past, files containing geographical data were transferred using magnetic tapes sent through the post. Since there are a variety of methods for storing data files on magnetic tapes, it was necessary to choose one that could be processed on both the *source* computer system from which the data originated and the *destination* computer system to which the data was moved. Sending magnetic tapes (and more recently CD-ROMs) through the post was also slow.

Use of magnetic tapes has been largely replaced by the use of communications networks for moving data. The wide-spread use of communications networks has greatly widened the range of data transfers that can be performed by interfaces for two reasons. First, the time taken to move data from one computer system has been greatly reduced, allowing data transfers to be performed more frequently. Second, the movement of data can be controlled by interfaces that execute on different computer systems.

To better understand the different kinds of data transfers that interfaces are expected to perform, the author divides any data transfer into a sequence of *transformations*. Each either changes the representation of data, or moves data from one computer system to another. A notation developed by the author for describing data transfers in terms of these transformations is presented in Chapter 3, and is used throughout the thesis to illustrate different aspects of data transfers and of the interfaces that perform those transfers.

Standardisation has become an important issue in the area of geographical data transfer. The focus of this effort has been the development of standard interchange formats such as the Spatial Data Transfer Standard (SDTS) (Geological Survey 1992) and the Map And Chart Data Interchange Format (MACDIF) (Evangalatos & Allam 1991). When a group of geographical information systems use a common data representation defined by the SDTS, for example, the number of separate interfaces required to transfer data among these geographical information systems is reduced.

Where an interchange format is used, data transfer between any pair of geographical systems requires two interfaces, one to transfer data from the source geographical information

system into the interchange format and another to transfer data from the interchange format into the representation required by the destination geographical information system. Because each interface can be used in many different transfers, this interfacing strategy is the most widely adopted of three strategies to be discussed in Chapter 3.

For both SDTS and MACDIF, parts of the data representation defined by these interchange formats are specified by international standards. The ISO standard 8211 (ISO8211 1985), which defines a data descriptive file for information interchange, forms part of the SDTS definition. The ISO standard 8824 (ISO8824 1987), which specifies a notation for defining data types, is used to define the types of data represented by MACDIF. SDTS and MACDIF are different in that the SDTS is oriented towards transferring data using files whereas MACDIF is oriented towards transferring data using communications networks. Use of these standards as part of the interchange formats SDTS and MACDIF is discussed in Chapter 4.

International telecommunications standards are becoming increasingly important as interfaces use communications networks to move data from one computer system to another. To distinguish interfaces that do move data from one computer system to another using a communications network, the author introduces in Chapter 4 the term *communicating interface* to refer to those interfaces that move data through a communications network. In this thesis, construction of communicating interfaces is considered within the framework of the ISO Reference model for Open Systems Interconnection (ISO7498 1984) and is assisted by using software tools and libraries provided by the ISO Development Environment (ISODE) (Rose, Onions & Robbins 1991).

Interface construction is a problem in many areas other than geographical information systems. Approaches to constructing interfaces for transferring databases, electronic manuscripts, and other application-related data are reviewed in Chapter 5. Reviewing these different approaches provides an opportunity to learn from experiences in interface construction for other fields, and to adapt any techniques that may be suitable for constructing interfaces to transfer geographical data. Approaches to constructing interfaces specifically for transferring geographical data are reviewed in Chapter 5.

The approach to interface construction explored in this thesis is that of generating interfaces from formal definitions of the data transfers. The approach is analogous to that which has become widely used in construction of compilers for programming languages, although the application area itself is very different. Each generated interface comprises a combination of interface modules, each of which performs one of the transformations necessary to transfer the data. In Chapter 6, four types of interface modules are defined, together with rules that determine how these modules are to be combined to form an interface.

Each transfer is formally defined by a transfer specification using a machine-readable notation that is described in Chapter 7. A *transfer specification* is divided into components, some of which may be used in several transfer specifications. Being able to reuse some components of a transfer specification reduces the time taken to define transfer specifications. These components are of three types: an interface definition, which specifies the sequence of data transformations needed to transfer the data; a representation definition, specifying different types of data and the method of storing these types of data values; and a translation definition, specifying the data transformations required to change the data from one representation to another.

Interfaces are generated from transfer specifications using `a2b`, a software tool designed and developed by the author to illustrate the approach described in this thesis. Interfaces generated by `a2b` are constructed using templates which contain components of an interface common to different transfers. Those components unique to a particular transfer are generated by `a2b` itself, or by `a2b` using software tools such as `bison` and `flex` and inserted into the interface templates. Techniques used to generate interfaces from transfer specifications are described in Chapters 8 to 11.

To investigate the concept of communicating interfaces and to develop techniques used in `a2b` for constructing this type of interface using ISODE, the author constructed the `gds/gdc` application, described in Chapter 12. This application contains many different communicating interfaces that can be used to transfer data to and from a variety of data representations. Data represented according to the data file formats Colourmap (CSIRONET 1986), Gina (GeoVision 1986), or Digital Line Graph (DLG) (Geological Survey 1990) can be transformed into data represented according to the Gina file format, for example, or data represented using the constructs of Postscript (Adobe 1985). Although constructed before `a2b`, the techniques used in the construction of `gds/gdc` are the same as those used by `a2b` to construct pairs of communicating interfaces. Examples of interfaces generated from transfer specifications using `a2b` are described in Chapter 13.

The description of the author's approach to interface construction is concluded in Chapter 14, with a discussion of ways in which the approach may be developed further as a consequence of experience in constructing `a2b` and using it to generate a range of interfaces. The author comments on the suitability of `a2b` for constructing not only interfaces to transfer geographical data, but interfaces to transfer *any* kind of data from one text-file representation to another.

Development of techniques to simplify interface construction, has led to a number of conclusions. Use of communication networks by interfaces has certainly changed the function and complexity of the interfaces that must be constructed. However, interface construction

has been simplified by providing a notation sufficiently powerful to specify a wide variety of transfers, which may include the use of communication networks, and a software tool that generates interfaces from these specifications.

Geographical data

Chapter 2

In this Chapter, concepts are introduced that are related to the digital representation and modelling of real world phenomena in any geographical information system. These concepts include:

a data model

which defines a notation for describing the types of data values, and operations for manipulating these data values;

an abstraction

which is a collection of ideas about the data to be represented; and

a schema

which is a definition of an abstraction using the notation and operations defined by a data model.

Digital representations for real world phenomena are determined by schemas defined at different levels of abstraction. In Section 2.2, three levels of abstraction are discussed: the conceptual level, the implementation level, and the physical level.

At each level of abstraction, a variety of schemas may be defined. At the highest level, diverse schemas may be defined to capture different aspects of the real world phenomena to be represented. A road, for example, may be viewed as either having several properties of interest such as the type of surface, the date last surfaced, and the number of lanes, or it may be viewed as one of many components of a census block boundary, in which case the previously mentioned properties of the road are of no interest.

At lower levels of abstraction, various schemas may be defined as a consequence of expressing the same abstraction using different notations and operations provided by different data models. Typically, a polygon may be represented using either the vector data model or

the tessellation data model. When using the vector model, the polygon is specified by the points which define the boundary of the polygon. When using a tessellation data model, all the space to be represented is divided into small units, and those units located within the polygon's boundary represent the polygon. Factors influencing the choice between the vector and tessellation data models are discussed in Section 2.1.

The difficulties encountered when transferring geographical data are due primarily to the diverse data representations defined for different geographical information systems. Examples of these different representations are described by Egenhofer & Herring (1991), Raper & Kelk (1991), and Mark (1978).

When a data set from a source geographical information system is transferred to a destination geographical information system, the data set will probably have to be transformed because the data sets at the source and destination geographical information systems conform to different conceptual, implementation, or physical schemas. A clear understanding of the source and destination schemas, and the data models that provide the underlying structure for the different schemas, will allow the concise specification of any data transformations to be performed during the transfer.

In this Chapter, the author also relates the concept of abstraction levels to 2 concepts used in the field of data communications for defining data transmitted through a network:

an abstract syntax
: which defines the types of data being sent through the network, and

a concrete syntax
: which defines a physical representation for values of the types defined by an abstract syntax.

Relating the concepts is important given the use of communication networks for moving data from a computer system to another system to be discussed in Chapter 4.

2.1 Data models

Tsichritzis & Lochovsky (1977) have defined a data model as 'a set of guidelines for the representation of the logical organisation of the data in a database ... (consisting) of named logical units of data and the relationships between them'. A more recent definition by Ullman (1988) states that 'a data model is a mathematical formalism with two parts:

1. A notation for describing data, and
2. A set of operations used to manipulate that data.'

The remainder of this Section contains brief descriptions of data models that will be used throughout this thesis.

2.1.1 The Entity Relationship model and its extensions

The Entity - Relationship model (ER) (Chen 1976), and subsequent extensions which are collectively referred to by Elmasri & Navathe (1989) as the Enhanced Entity Relationship model (EER), are widely used models for defining data abstractions, although they do not fully conform to Ullman's definition of a data model because neither include any operations for manipulating this data.

The ER model provides three basic concepts:

an entity type
: which is an abstraction of any distinguishable real world phenomenon;

an attribute
: which is a property of an entity type, and whose value describes a particular instance of an entity;

a relationship type
: which describes some form of association between entities. Martin & McClure (1985) suggest four types of relationships:

  Basic
  : which indicates how many instances of one entity type can be related with another entity (1 to 1, 1 to many, many to many),

  Labelled
  : which is a basic relationship with text that describes what the relationship represents,

  Looped
  : which indicates a basic, or labelled relationship within the same entity type,

  Linked
  : which indicates that there is some type of connection among the relationships made. For example, relationships can be linked to indicate that instances of the entity types involved with the linked relationships will occur only when all of the linked relationships are true.

A particular virtue of the ER model is that there are simple techniques for expressing schemas as ER diagrams. At least three types of representation are used for ER diagrams

(Martin & McClure 1985): Crow's-foot notation (used in this thesis), Arrow notation, and Bachman notation.

The ER model was designed for expressing business-oriented conceptual schemas. Extensions to the ER model were devised because of its limitations in expressing schemas for applications such as those developed in engineering, graphics, and geography. These extensions include: generalisation, specialisation, aggregation, and association.

Generalisation refers to the viewing of 'a set of similar objects ... as a generic object' (Smith & Smith 1977). The generic object is regarded as an entity super-class and the set of similar objects are regarded as being sub-classes of this super-class. The super-class comprises attributes common to all the sub-classes. Any sub-class inherits the attributes defined for its super-class, and may define its own collection of attributes.

An example of generalisation given by Worboys, Hearnshaw & Maguire (1990) is that the entity types `village`, `town`, and `city` may be generalised to form the entity super-class `settlement`. The idea of classes and sub-classes with attribute inheritance raises an interesting possibility for spatial data. For a geographical information system, there are a few spatial entity types such as point, line, and polygon which are used to represent the spatial component of real world phenomena. In effect, each spatial entity type is a generalisation of many different entity types representing various real world phenomena.

For example, the entity types representing census, political, and statistical boundaries may be generalised to form the entity type `polygon`. The entity super-class `polygon` has an attribute that describes the spatial location of either census, political or statistical boundaries, and the entity sub-classes `census boundary`, `political boundary`, and `statistical boundary` having attributes describing aspects unique to the corresponding real world phenomena.

Specialisation is the process of defining a collection of entity sub-classes based upon a single entity type which becomes the super-class. An entity type such as `vehicle` may form the basis for defining specialised classes such as `cars` and `trucks`.

Aggregation is the construction of a single higher-level entity type from a number of different entity types. An example given by Smith & Smith (1977) is the creation of an entity type `reservation` for describing the relationship among the entity types `person`, `hotel`, and `date`. Another example, given by Worboys *et al* (1990) is the aggregation of the entity types `point identifier`, `x coordinate`, and `y coordinate` to form the entity type `point`.

Association allows entities of the same type to be grouped together to form another entity type. An example given by Worboys *et al* (1990) is the grouping together of entities representing districts to form an entity type `city`. When necessary, the entities forming an

R

supplier	*part*
1	1
2	2
2	4

S

part	*project*
1	1
1	2
2	1

P

part	*colour*
1	blue
2	red
3	green

(a) Three relations

T

supplier	*part*	*project*
1	1	1
1	1	2
2	2	1

(b) The natural join of relation R with relation S

R'

supplier	*part*
1	1
2	2

(c) The restriction of relation R by relation P ($R.part = P.part$)

Figure 2.1: Examples of relations and relational operators

association may be ordered. For example (*op cit*), a `road` entity type may be defined by an ordered group of entities representing intersections.

2.1.2 Relational model

The relational data model (Codd 1970) defines the concepts that have been widely implemented as relational data base management systems such as Ingres and Oracle. The basic concept of the relational model is that data should be represented in a collection of relations (essentially tables), each containing a number of tuples (rows) with values of attributes (columns). Examples of relations are shown in Figure 2.1(a). Operations defined by the model for manipulating relations include:

Selection

The selection of one relation produces another that is either or both a permutation, or a projection of the original relation. A permutation of one relation produces another with a different ordering of the attributes constituting the original relation. A projection of some relation will produce another with a subset of attributes from the original relation.

Join

A variety of join operations are defined (see (Codd 1970) for a complete description).

The natural join of two relations R and S with a common attribute produces a third relation T consisting of all attributes from the relations R and S (see Figure 2.1(b)). Tuples of the new relation T are formed by taking values from tuples of the relations R and S that have the same value for the common attribute.

Restriction

Any subset of tuples contained within some relation R may form a new relation R'. Tuples in relation R' are restricted to those in relation R that have attribute values equal to attribute values of tuples in some third relation P. In Figure 2.1(c), for example, restricting relation R by relation P produces relation R'.

Although a relational data base is generally ideal for representing non-spatial data, representing spatial data within a relational data base is more difficult. These difficulties occur because, on the one hand, no ordering of tuples within a relation is allowed and, on the other, much spatial data is inherently ordered. Examples of this ordering include: the sequence of points defining a line, and the sequence of lines that describe a clockwise traversal of some polygon's boundary. In the spatial representation defined by van Roessel (1987), relations include a column for storing sequence numbers to specify the ordering of the represented data.

One approach to overcoming this limitation of the relational model is to allow the columns of relations to store more complex types of data. In Figure 2.2(a), a representation of lines is shown using the original definition of the relational model, with sequence numbers used to preserve the order of points defining the line. In Figure 2.2(b), the representation of the same lines is shown using a variant of the relational data model defined for the Postgres data base management system (StoneBraker 1992), with a collection of points allowed to be stored as one data value.

2.1.3 Object-oriented models

The term *object-oriented* has been used in many fields of computer science including programming languages, databases, artificial intelligence, and computer science systems in general (Elmasri & Navathe 1989). Various object-oriented data models have been developed in these fields, but no definitive object-oriented model has been generally accepted. Elmasri & Navathe (1989) describe four 'essential features' of these different object-oriented models:

data abstraction and encapsulation

An object class (or object type) is a data abstraction similar to an entity type of the Entity Relationship model. An object class is an encapsulation or grouping of attributes

R

Line Id	*Sequence*	*X*	*Y*
⋮	⋮	⋮	⋮
1049	0	10	3
1049	1	15	5
1050	0	12	17
1050	1	17	20
1050	2	21	25
⋮	⋮	⋮	⋮

(a) Representation of lines using a traditional relation

R'

Line Id	*points*
⋮	⋮
1049	(10, 3), (15, 5)
1050	(12, 17), (17, 20), (21, 25)
⋮	⋮

(b) Representation of lines using a Postgres relation

Figure 2.2: Representations of lines within a traditional relation and a Postgres relation

that describe an object, and the operations or *methods* which provide the only way in which a user may access or modify attribute values of an object.

inheritance

Object classes may be organised into *type hierarchies*, analogous to the specialisation and generalisation hierarchies that may be defined using the Enhanced Entity Relationship model described in Section 2.1.1. Any definition of an object sub-class may include some or all of the attributes and methods of the object's super-class.

object identity

Any object has an identity that is independent of the object's attribute values. This allows any or all of the attribute values to be modified without losing the identity of the object.

complex objects

A class of *complex objects* defines a type of object that consists of some combination of other object classes.

Object-oriented data models have been found useful in defining representations for geographical data (see, for example, Worboys *et al* (1990), Milne, Milton & Smith (to appear), Kemp (1990)). Worboys *et al* (1990) suggest that object-oriented data models reduce the gap that exists between 'the richness of the knowledge structures in the application domains and the relative simplicity of the data model in which these structures can be expressed and manipulated.' They use a variant of the object-oriented data model called IFO

(Is-a relationships, Functional relationships, complex Objects) to describe some standard cartographic primitives, such as node, line and polygon, and some standard administrative units in the United Kingdom. The concepts of an object-oriented model were also used by Worboys (1992) to define the geometric or spatial data model described in the next Section.

2.1.4 Geometric models

Spatial or geometric data models define methods for dividing continuous real world data into discrete units of space that can be digitally represented. Goodchild (1992) refers to the transforming of continuous real world data into discrete digital data as '*discretization*'. Many informal geometric data models have been defined. Briefly, these models are (Peuquet 1984):

vector models

in which spatial data objects are defined using either points specified by coordinates, or other spatial objects that are themselves defined using points. Vector models may be further divided into two groups:

spaghetti models

where objects are defined without representing any topological relationships such as adjacency, and

topological models

where the definition of objects includes a representation for topological relationships.

tessellation models

in which the domain of the spatial objects is divided into units of space, groups of which are then used to represent spatial objects. Peuquet identifies five variations of the tessellation model:

regular tessellations

based around the three geometries of square, triangular, and hexagonal meshes.

nested tessellations

based on subdividing the elemental polygon of the grid into polygons of the same shape. The most recognised of these models is the quad-tree (Samet 1984).

irregular tessellations

which differ from regular tessellations as the elemental polygons within a mesh

may vary in size. The most commonly used model of this form is called the triangulated irregular network (TIN) (Peucker, Fowler, Little & Mark 1978).

scan-line tessellations

which are a special case of the square mesh where the cells are organised into single, contiguous rows across the data surface, usually parallel to the x axis.

Peano scan tessellations

which are based on mappings of n-dimensional space onto lines and vice versa. Peano scans are found to be useful for image processing applications (Stevens, Lehar & Preston 1983).

Peuquet notes that:

> 'tessellation or polygonal mesh models, represent the logical dual of the vector approach. Individual entities become the basic data units for which spatial information is explicitly recorded in vector models. With tessellation models, on the other hand, the basic data unit is a unit of space for which entity information is explicitly recorded.'

The vector and tessellation data models have been widely discussed and compared in the literature (see, for example, Maffini (1987), Holroyd & Bell (1992)). An example of how a polygon may be represented, by either the vector or tessellation data models, is shown in Figure 2.3.

Choosing between vector and tessellation data models is usually decided according to the method used to capture data, the nature of the data being represented, and the expected types of data processing. Digitising data from existing maps produces data values based on the vector data model (points, lines, and polygons). Image processing and remote sensing produce data values based on tessellation data models such as rasters and quad-trees.

The vector-based models are typically used when the spatial data is more naturally represented by lines and polygons (and perhaps topology is required). Examples of such data are:

cadastral data

which describe property and other legal boundaries;

administrative and political data

which describe the boundaries of national and local regions of authority;

statistical data

which describe the boundaries of regions about which information is gathered;

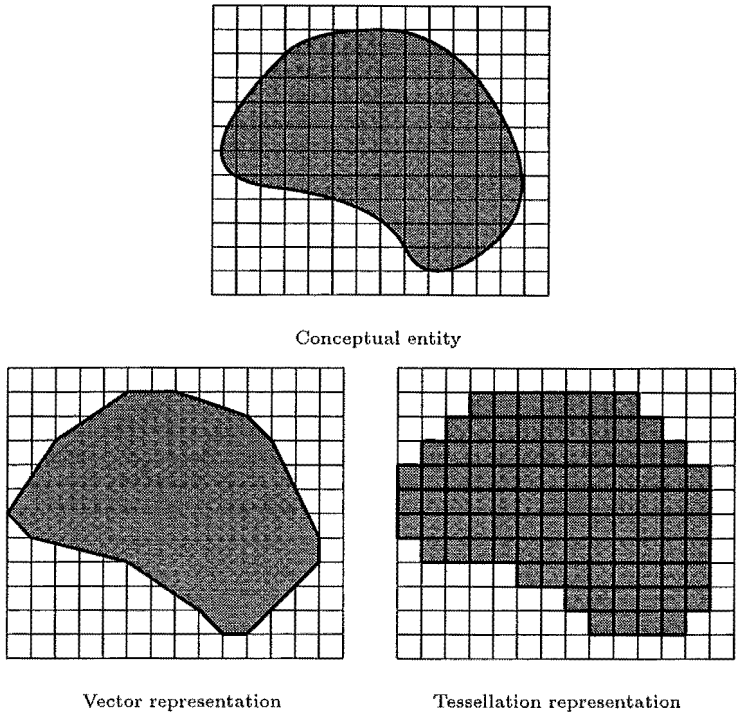


Figure 2.3: Data representation using either a vector model or a tessellation model

transportation data

which describe the location and topology of roads, railways, shipping lanes, and airways; and

service networks

which describe the location of utilities such as drainage, electricity, and telecommunication lines.

Tessellation data models are used to represent images captured by satellite, or for data captured by sampling the real world. Examples of data often represented using tessellation models are: types of vegetation or soil, and elevation data.

These informal geometric data models provide constructs for representing data, without providing a formal definition of a notation or the operations for describing and manipulating the represented data as suggested by Ullman (1988). Worboys (1992) notes that there is 'a lack of a coherent theory underpinning geographical databases'.

Worboys (1992) has therefore proposed a generic model for planar geographical objects, intended to 'provide a partial theory, namely the formalisation of the underlying object model

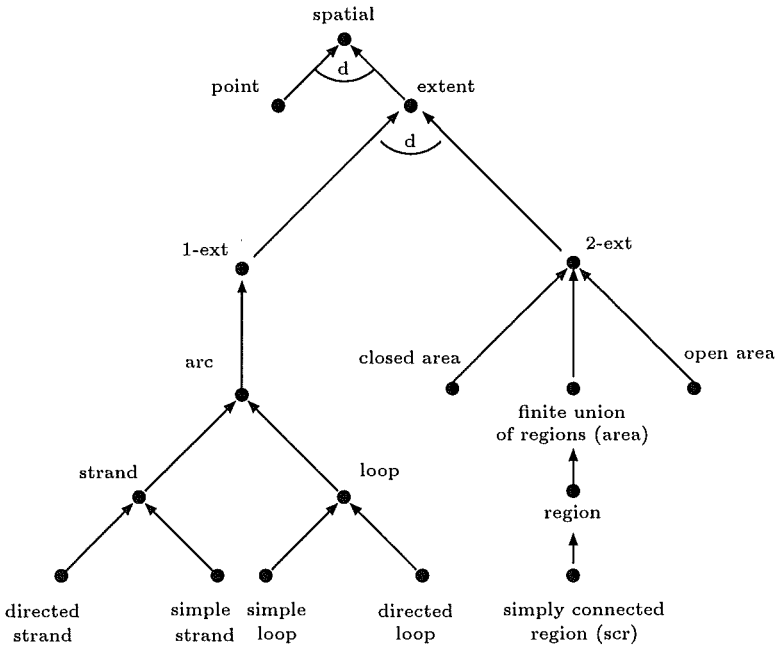


Figure 2.4: An inheritance hierarchy of spatial object classes (Worboys 1992)

for geographical data whose spatial references are embedded in the plane'. This theory of spatial objects is designed using the general principles defined by the object-oriented data model described in Section 2.1.3. The generic model proposed by Worboys consists of three parts:

1. object model in continuous space
2. object model in discrete space
3. object model in representation space

The object model in continuous space consists of an inheritance hierarchy of spatial object classes (reproduced in Figure 2.4), and a set of operations defined upon these spatial object classes. These objects and operations in continuous space are then mapped to a set of objects and operations in discrete space. The discrete spatial objects are: points, piecewise linear arcs, and polygonal areas (hence the mapping is a vector discretization). The object models in continuous and discrete space may be thought of as 'two many-sorted algebras, each equipped with a set of sorts of objects and a set of operations which act on objects of the correct sort'.

The method of representation for spatial objects in discrete space uses the notion of a 'simplicial complex' from combinatorial topology, and is an ongoing research project (Worboys 1992). Use of simplicial complexes for studying spatial relations was described earlier by Egenhofer, Frank & Jackson (1989).

Formal definition of the geometric models of geographical information systems would greatly assist the construction of interfaces. Comparing these definitions is expected to show the data transformations necessary to transfer data from one geographical information system to another. Without these definitions, interface construction is a process of trial and error.

2.2 Abstractions

Data models define the concepts and operations which are used at a variety of levels to specify abstractions of the real world phenomena. High-level abstractions describe the real world phenomena in terms that are closest to the ways which people might use most naturally. Lower-level abstractions describe higher-level abstractions in terms that are more suitable for computers, with the lowest level being expressed in terms of computer storage structures.

Three levels of abstraction are defined here:

the conceptual level

at which the relevant properties and relationships of the real world phenomena are defined, without considering the method of storing these properties and relationships in a computer;

the implementation level

at which the properties and relationships defined at the conceptual level are expressed using the data types and operations defined by one or more *implementation data models*. An implementation model defines the concepts and operations implemented as software systems. The relational model, for example, defines the fundamental concepts implemented within a relational database management system; and

the physical level

at which the data types defined by an implementation data model are expressed using the definition of data structures and algorithms provided by physical data models for representing and manipulating data values. Physical data models comprise data structures and operations which are typically defined using pseudo-code.

A conceptual abstraction is defined by a schema that is usually expressed using the Entity-Relationship model described in Section 2.1.1. An implementation abstraction is defined by

a schema using the Data Definition Language provided by a database management system. The database management system is based on an implementation model such as the relational or an object-oriented data model. A physical abstraction is defined by a schema that specifies the memory and disk data structures, typically using one or more programming languages.

Sections 2.2.1, 2.2.2 and 2.2.3 are detailed discussions of these abstraction levels, and contain examples of schemas at different levels of abstraction. Section 2.2.4 is a comparison between the framework provided by the conceptual, implementation, and physical levels of abstraction and other frameworks which have been proposed.

2.2.1 Conceptual

For a geographical information system, a conceptual abstraction specifies the relevant properties and relationships of real world phenomena to be digitally represented, in terms that those people using the data might find most natural. Different types of conceptual abstractions can be defined to correspond to different types of map. For example, topographical and thematic schemas define the content of topographical and thematic maps.

The content of a simple cadastral map is described by a cadastral schema that defines an abstraction comprising two entities:

parcels

the smallest legal unit of land with characteristics such as: the legal parcel name; the name of the owner; the residential address; the parcel's location; and the area of the parcel; and

parcel owners

with characteristics such as: the owner's name and the owner's residential address.

The cadastral schema may also define constraints and relationships between the parcels and owners. Suppose that, for some cadastral application, the following relationships and constraints are defined:

- An owner relationship such that: a parcel belongs to a single owner; and that an owner must own one or more parcels;
- An integrity constraint where the name of a parcel owner, which is a characteristic of the parcel abstraction, must be a reference to an instance of the owner abstraction; and
- A topological relationship such that a parcel may be adjacent to one or more other parcels. Thus, a parcel may have one or more boundaries in common with other parcels.

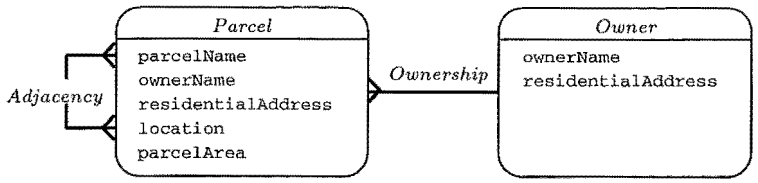


Figure 2.5: A conceptual schema expressed using an Entity - Relationship diagram

Such conceptual abstractions may be defined by a conceptual schema, which is expressed using the notation and operations defined by a conceptual data model. A conceptual schema is often expressed using either the Entity-Relationship (ER) model (Chen 1976) or the Enhanced Entity Relationship (EER) model (Elmasri & Navathe 1989) described in Section 2.1.1. The conceptual schema for the cadastral data defined above is expressed in Figure 2.5 as an ER diagram.

A conceptual schema is a detailed description of the data types and relationships that are to be represented in a computer, without having to describe the computer representation itself. The conceptual schema is therefore expressed in a form that can be designed and discussed by users familiar with the real world phenomena, but not with representations of data within a computer. Consideration of the digital representation of this data is first given while defining an implementation abstraction.

2.2.2 Implementation

An implementation schema defines an abstraction equivalent to that of the conceptual schema using the concepts provided by an implementation model. Most early geographical information systems automated the editing and plotting of maps. Consequently, the implementation models of these geographical information systems were almost entirely spatial data models such as those described in Section 2.1.4. The need to store additional non-spatial information about these maps led to the linking of some form of database management system (DBMS) with the software for editing and plotting maps. Such a geographical information system is based on an implementation model consisting of two parts:

spatial model

which defines the geometric and topological constructs for representing the spatial aspects of real world phenomena, and a

non-spatial model

which defines the constructs for storing and processing the non-spatial aspects of real

world phenomena. Of the hierarchical, network, and relational data models developed in the area of DBMSs, the relational model, described in Section 2.1.2, is the most frequently used.

ARC/INFO is an example of such a geographical information system consisting of two parts: ARC, which provides support for the editing, plotting, and analysis of spatial data; and INFO, which is a DBMS used for the storing and processing of non-spatial data.

Recently, the storing and processing of both spatial and non-spatial data within one system has been achieved by extending relational DBMSs, or by using object-oriented DBMSs. A future version of the Ingres relational DBMS will support multi-dimensional data to support application areas such as geographical information systems (Ingres 1993). The use of the object-oriented DBMS ONTOS for storing and processing geographical data is discussed by Milne *et al* (to appear). Postgres (StoneBraker 1992) is an experimental DBMS that uses both the relational model, extended to allow the definition of more complex types of attributes, and object-oriented concepts such as inheritance to support the storage and retrieval of highly structured data.

2.2.3 Physical

At the physical level of abstraction, data representations are specified in terms of data structures and algorithms for manipulating and traversing these structures. A geographical information system is an implementation of these structures and algorithms. Therefore, those who design and construct a geographical information system define the physical schema using the constructs provided by some programming language.

A user 'chooses' the physical representation of data either by selecting one of many geographical information systems that provide different representations, or by selecting one of many representations provided by one geographical information system. A user cannot define the physical representation of data, unless the person is involved with the design or construction of a geographical information system.

There is a huge variety of physical data models which define representations for data values in computer memory and magnetic tape or disk. Many of the well-known methods are discussed by Samet (1990). Of interest here are those physical representations used for data values transferred from a source geographical information system to a destination geographical information system. In this thesis, these physical representations will be called *transfer representations*.

2.2.3.1 Transfer representations

Data values have been traditionally transferred in files stored on magnetic tape. However, data values are now being transferred as memory-resident values between applications linked by a communications network. For the purpose of this thesis, transfer representations are therefore divided into two classes:

representations for memory-based transfers

methods of representing values to be transferred directly from a source application to a destination application through a communications network. Definition of network representations is discussed further in Section 2.3;

representations for file-based transfers

methods of representing values in either a binary file, or a text file.

Binary files are more compact data representations than those of text files, but there are many compression techniques that may be used to reduce the size of any text file. A disadvantage of any binary file representation is that values have to be accessed using applications designed only to read and write this binary representation. In contrast, data values represented in a text file may be viewed and modified using any text file editor, as well as any application constructed for reading and writing this representation.

Text file representations are essentially sequences of characters that may be divided into groups of characters called *tokens*. These tokens may themselves represent values, or be combined to represent more complex values. In Figure 2.6, for example, the sequence of characters `'142.782,45.672'` is divided into two real tokens `'142.782'` and `'45.672'`, and a separator `','`. The two real tokens represent longitude and latitude values which are combined to represent the value of a point.

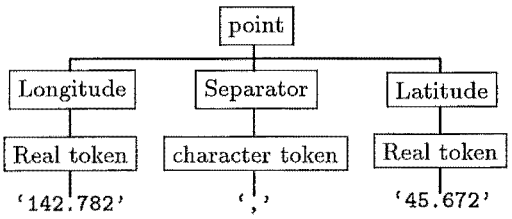


Figure 2.6: A text file representation of a point

Viewing text file representations as sequences of characters or tokens leads to the use of grammars for specifying the representation. Grammars are frequently used in the field

of computer languages and compiler construction, and in later Chapters the analogy will be drawn between parsing a program written in some computer language and parsing a text file representation of geographical data. Software tools for assisting the construction of compilers have been used by the author for assisting the construction of software for reading and writing text file representations.

A text file representation typically consists of tokens that are either fixed or variable in size. Any fixed size token consists of a set number of characters. In the case of the Colourmap file format (CSIRONET 1986), for example, tokens that represent real and integer values must consist of 10 characters, while tokens representing string values must, in one case, consist of 80 characters (a comment describing the data represented within the file), and, in another, consist of 30 characters (an attribute description). When a value requires fewer characters, the blank character ' ' is used to pad out the token.

Representations with variable-sized tokens rely on the use of special tokens, often called *delimiters*, to indicate the end of one token and the start of the next. In Figure 2.6, for example, the character token ',' separates the two real tokens '142.782' and '45.672'. The GeoVision Gina file format (GeoVision 1986), uses delimiters such as 'udb-feature', 'feat', ',', and ' ' (a space) as shown in Figure 2.7.

```
udb-start b v 8192
⋮
udb-primary
road free
field name char 40 i n "road name"
field rtype char 12 i n "type of road:  arterial; highway; etc."
field lanes num 2,0 "maximum number of lanes"
field surface char 10 n "type of surface"
fc 1000,1099
⋮
udb-feature
feat 1 1002 2 0 1 xy 0 0 0 1 0
coor 1805 1120 1805 1143
coor 2018 1143 2018 1204 2508 1204
label 1 1805 1120 "horizontal" 90.
text "Elm Street"
attr Elm suburban 2 "asphalt"
⋮
udb-end
```

Figure 2.7: An example of a self-defining text file representation based on that given in GeoVision (1986)

Some text file representations are *self-defining*. That is, some data representations are both defined and used within the same text file. An example of a text file representation which

is in parts self-defining is the GeoVision Gina file format (GeoVision 1986). In Figure 2.7, a table called `road` is defined with four attributes `name`, `rtype`, `lanes`, and `surface`. Features with a code in the range of 1000,1099 may have an associated attribute record with values for these attributes. The feature shown in Figure 2.7 has a code of `1002`, consequently, the associated attribute record describes this feature as a `suburban` road called `Elm` with a maximum of 2 lanes and an `asphalt` surface.

2.2.4 Comparison with other abstraction frameworks

The conceptual, implementation, and physical levels of abstraction provide a framework that is similar to many others described in the literature (see, for example, Frank (1992), Goodchild (1992), the ANSI/SPARC architecture (Tsichritzis & Klugs 1978), and the architecture used for the Nagiya Project (Gawkowski & Mamrak 1992) to be discussed briefly in Section 5.1.4). A comparison is now made of the conceptual, implementation, and physical levels of abstraction defined above, and two other frameworks, the ANSI/SPARC architecture (Tsichritzis & Klugs 1978), and the framework discussed by Frank (1992).

The ANSI/SPARC architecture for database systems is frequently referred to in the literature (see, for example, Elmasri & Navathe (1989), Ullman (1988), and Date (1990)). The ANSI/SPARC architecture was proposed (Tsichritzis & Klugs 1978) to assist in insulating programs from data, supporting multiple user views, and storing the description of the database in a catalog (Elmasri & Navathe 1989). As described by Date (1990), the ANSI/SPARC architecture is divided into three levels:

- The *internal* level is the one closest to physical storage – it is concerned with the way the data is physically stored;
- The *external* level is the one closest to the users – it is concerned with the way the data is viewed by individual users; and
- The *conceptual* level is a 'level of indirection' between the other two.

Date observes: 'there will be many distinct "external views", each consisting of a more or less abstract representation of some portion of the total database, and there will be precisely one "conceptual view", consisting of a similarly abstract representation of the database in its entirety. Likewise, there will be precisely one "internal view", representing the total database as physically stored.'

Although the internal level of the ANSI/SPARC architecture and the physical level of abstraction described in Section 2.2.3 are equivalent, the external and conceptual levels of

the ANSI/SPARC architecture are different from the conceptual and implementation levels of abstraction discussed in Sections 2.2.1 and 2.2.2 respectively. In the ANSI/SPARC architecture, the distinguishing characteristic between the external and conceptual levels is whether an abstraction is being defined to reflect an individual user's perspective of some or all of the database, or whether an abstraction is being defined for the entire database. In contrast, the distinguishing characteristic between the conceptual and implementation levels of abstraction is whether the abstraction is being defined independently of the method used to represent values of the data types defined by the abstraction, or the abstraction is being defined in terms of what is provided by an actual geographical information system.

The framework proposed by Frank (1992) is specific to the representation of geographical data. Frank's framework is based on three concepts:

spatial concepts and geometry

'ideas, notions and relations between them which are used by humans to organize and structure their perception of reality'. This concept corresponds to those defined at the conceptual level of abstraction defined here;

a geometric data model

'a formalized abstract set of spatial object classes and the operations performed on them', which corresponds to those defined at the implementation level of abstraction; and

geometric data structures

'the specific implementation of a geometric data model, which fixes the storage structure, utilization and performance', which corresponds to those defined at the physical level of abstraction.

The concept of abstraction levels is discussed below in the context of concrete and abstract syntaxes, as used in the field of communication networks for data representation.

2.3 Abstract, concrete, and transfer syntaxes

Using communication networks to move data from one computer system to another has led the author to associate the concept of abstraction levels with the concepts of concrete and abstract syntaxes as used in the field of communication networks for data representation.

An *abstract syntax* is 'the specification of Application Layer data ... using notation rules which are independent of the encoding technique used to represent them' (ISO8822 1988).

Therefore, an abstract syntax is a definition of data types without defining the physical representation for values of these types.

The physical representation of different types of data values is defined by a *concrete syntax*, which specifies 'those aspects of ... the formal specification of data which embodies a specific representation of that data' (ISO7498 1984). In this thesis, a concrete syntax specifies the method of representing data values within computer storage. This specification may be in terms of the structures used within data files on disk, memory-resident data structures provided by some programming language, or some form of electronic representation through a communications network.

A concrete syntax, which defines some form of electronic representation of data values to be used by two communicating applications, is referred to in this thesis as a *transfer syntax*. In the ISO-OSI standards, a transfer syntax is defined to be 'that concrete syntax used in the transfer of data between open systems' (ISO7498 1984). For any transfer of values from one application to another through a network, there will be three concrete syntaxes: one local to each of the applications, and the transfer syntax.

Broadly speaking, conceptual and implementation level schemas correspond to two abstract syntaxes. One syntax, corresponding to the conceptual schema, defines the conceptual abstraction as a collection of data types expressed using predefined general data types such as integer, boolean, and string. A second syntax, corresponding to the implementation schema, defines the same conceptual abstraction using data types specific to some geographical information system. The physical schema corresponds to a concrete syntax. Both the schema and the syntax define physical representations for data values, of the types specified by the implementation schema or the corresponding abstract syntax, in terms of computer storage structures.

Using the concepts of abstraction levels on the one hand, and abstract syntaxes and concrete syntaxes on the other, the process of specifying a digital representation for geographical data can be thought of as:

1. specifying a conceptual abstraction and a corresponding abstract syntax for the real world phenomena to be represented;

2. specifying an equivalent implementation abstraction and a corresponding abstract syntax using the data types and operations provided by the particular geographical information system to be used for representing the data; and

3. using the physical schema and a corresponding concrete syntax for the data values defined during the construction of the geographical information system. Usually, neither

the physical schema nor the corresponding concrete syntax can be defined by the user of a geographical information system.

Constructing interfaces to perform data transfers requires an understanding of the differences between various representations of geographical data. These differences determine the various data transformations that must be performed by interfaces during the transfer. In the next Chapter, a variety of data transfers is discussed in terms of the different data transformations that have to be performed.

Data transfer

Chapter 3

Traditionally, data files were moved from one computer system to another using magnetic tapes sent through the post. Recently, the widespread use of communication networks has both reduced the time taken to move data and significantly increased the range of data transfers requested by users of geographical information systems. These changes to the transfer environment are discussed in Section 3.1.

The process of transferring data has become more complex as a consequence of the greater access to data by the use of communication networks. A notation, presented in Section 3.2, is used extensively throughout this thesis for describing different types of data transfer and for describing various elements which constitute a data transfer.

Data may be lost in the transfer for many reasons which are described in Section 3.2.1. In some cases, data is lost because data representable in the source geographical information system cannot be represented in the destination geographical information system. In others, data is lost because of the process used for transferring the data.

Technological developments allow users to transfer data in a much greater range of different environments which are discussed in Section 3.1. Data transfers for each of these environments have different structures which are described in Section 3.3. Section 3.4 is a comparison of different strategies for governing the use of interfaces to transfer data among many geographical information systems. The Chapter is concluded by expressing the primary objective pursued in this thesis, to develop simpler and more systematic methods for constructing interfaces.

3.1 Transfer environments

Different transfer environments are discussed in this Section beginning with the use of CD-ROMs, which are replacing magnetic tapes for bulk data transfer. CD-ROMs are a new form of disk which provide a portable storage medium on which large volumes of data can be stored. Data stored on a CD-ROM may be accessed quicker than from a magnetic tape because CD-ROM drives provide direct access to the stored data, unlike magnetic tape drives which provide sequential access.

The concrete representation of data stored on a CD-ROM is controlled by formal definitions or standards at both the hardware and the software level. This removes the often awkward problems resulting from the many diverse tape formats which are informally defined and may be computer system-dependent.

Moving data from one system to another using magnetic tapes was traditionally a time-consuming process because of the delay between sending and receiving these tapes. Use of CD-ROM does nothing to improve the time taken since a CD-ROM is still posted from one system to another. An improvement in this aspect of data transfers is accomplished using computer networks.

Use of communication networks both speeds up the movement of data, and provides a greater variety of methods for accessing data stored on a remote computer system. More varied and complex data transfers are now possible as a consequence of these different methods of accessing remote data. For example:

- the use of a *file server* for sharing files among a group of computer systems, which are usually located within one building and connected by a local area network (LAN);
- the use of *electronic mail* for the world-wide communication of mail among users and, more recently, for the distribution of data files by electronic mail servers;
- the development of protocols and applications such as FTAM and ftp for transferring data files from one system to another through a communications network;
- the use of hand-held collectors for recording field notes which are transferred through a specialised, dedicated communication link;
- the development of distributed database systems for sharing data among users on many computer systems; and
- the development of applications which, although executing on different systems, communicate with each other through a network.

Many of the current geographical information systems are analogous to file servers in that these systems enable many users to simultaneously browse the same maps. It seems reasonable to expect geographic information systems to develop into distributed geographical information systems, in the same way that centralised database systems have tended to develop into distributed database systems. A distributed database system manages data which may be stored on several computer systems connected by a communications network.

As an example, consider that the relational database management system Ingres has evolved into INGRES/STAR which, when used with INGRES Gateways, allows users to access 'both INGRES and non-INGRES databases transparently and simultaneously in an integrated distributed heterogeneous environment' (Ingres 1989). In association with a developer of spatial data management applications, Ingres has also recently extended the capabilities of the system to allow the management of multidimensional data which form maps (Ingres 1993).

The variety of methods available for accessing data greatly widens the scope for transferring geographical data from one geographical information system to another. A notation to be frequently used in this thesis is presented below for describing the transfer of geographical and other kinds of data.

3.2 The transfer process

Symbols shown in Table 3.1 are used to describe the different elements of a data transfer, and are combined to form descriptions of different data transfers. A description of these symbols is given below, together with an explanation of how they are combined to describe different data transfers.

The transfer of data from a geographical information system to another geographical information system is accomplished by a sequence of transformations. A data transformation modifies data values to conform to a different property. Properties associated with data values are:

- a formal definition comprising a conceptual schema C, an implementation schema I, and a physical schema P, and
- a location where the data set is stored. The data set will be typically stored at either a source computer system, L_S, or at a destination system L_D.

The properties of a set of data values $\mathcal{A}$ can be described at a given time by:

$$\mathcal{A}(C_A,\ I_A,\ P_A,\ L_A)$$

Notation	Associated term	Definition
C	A conceptual schema	An expression of a conceptual abstraction specifying the relevant properties and relationships of real world phenomena to be digitally represented.
I	An implementation schema	An expression equivalent to the conceptual schema, but using the geometric, topological, and non-spatial concepts of an actual geographical information system.
P	A physical schema	An expression of the structures and algorithms for digitally representing and manipulating data values, using the constructs provided by a programming language.
L	Location of a data set	The computer system at which the data values are stored.
$\mathcal{A}(C_A,\ I_A,\ P_A,\ L_A)$	A data set	The set of data values $\mathcal{A}$ conforming to the schemas C_A, I_A, and P_A, and stored at location L_A. Different types of data set are shown by replacing the symbol $\mathcal{A}$ with: $\mathcal{S}$ a *source* data set — a set of data values stored by the geographical information system *from* which data is to be transferred. This set conforms to the schemas C_S, I_S, and P_S of that geographical information system. $\mathcal{D}$ a *destination* data set — a set of data values stored by the geographical information system *to* which data is to be transferred. This set conforms to the schemas C_D, I_D, and P_D of that geographical information system. $\mathcal{T}$ a *temporary* data set — a set of data values that may occur *during* a data transfer. This set conforms to a representation that may be of interest only to those constructing or maintaining software to perform data transfers. $\mathcal{I}$ an *interchange* data set — a set of data values conforming to the schemas C_I, I_I, and P_I of standards that define some interchange representation. $\mathcal{N}$ a *network* data set — a set of data values conforming to the schemas C_N, I_N, and P_N defined to give a common representation for data to be transferred across a network.
$\mapsto$	A data *transfer*	The transfer of a data set from one representation to another.
$\rightarrow$	A data *translation*	The transformation of a data set to conform to a different conceptual, implementation, or physical schema.
$\rightsquigarrow$	A data *movement*	The movement of a set of values from computer system x to computer system y.
$\overset{*}{\rightarrow}$	Many data transformations	One or more data translations or data movements.

Table 3.1: An explanation of the notation for describing the process of transferring data

The set of data values $\mathcal{S}$ to be transferred will conform to a representation defined by the conceptual, implementation, and physical schemas C_S, I_S, and P_S, and will be located on the computer system L_S. This source data set is described by:

$$\mathcal{S}(C_S,\ I_S,\ P_S,\ L_S).$$

After transfer, the data set is denoted by:

$$\mathcal{D}(C_D,\ I_D,\ P_D,\ L_D).$$

Thus, the aim for a data transfer is to transform data set $\mathcal{S}$ conforming to a representation defined by C_S, I_S, P_S, and located on system L_S, into the other data set $\mathcal{D}$ conforming to a representation defined by C_D, I_D, P_D, and located on system L_D. This transfer may be described using the symbol $\mapsto$ as follows:

$$\mathcal{S}(C_S,\ I_S,\ P_S,\ L_S) \mapsto \mathcal{D}(C_D,\ I_D,\ P_D,\ L_D)$$

which may be abbreviated to $\mathcal{S} \mapsto \mathcal{D}$. When the data values are located on the same computer system throughout the transfer, the location property is dropped from the description. Thus, a transfer may be described as: $\mathcal{S}(C_S,\ I_S,\ P_S) \mapsto \mathcal{D}(C_D,\ I_D,\ P_D)$.

During the transfer, the data values may be transformed into one or more temporary data sets $\mathcal{T}_i$ having different groups of properties. For example, a transformation applied to data set $\mathcal{T}_j$ may change the values to conform to a different physical schema. Such a transformation is described as:

$$\mathcal{T}_j(C_j,\ I_j,\ P_j) \rightarrow \mathcal{T}_k(C_j,\ I_j,\ P_k).$$

A data transfer $\mathcal{S} \mapsto \mathcal{D}$ comprising several transformations is described as:

$$\mathcal{S} \rightarrow \mathcal{T}_1 \stackrel{*}{\rightarrow} \mathcal{T}_j \rightarrow \mathcal{T}_{j+1} \stackrel{*}{\rightarrow} \mathcal{T}_k \rightarrow \mathcal{D}$$

where the data values are transformed through k temporary forms, and the temporary set $\mathcal{T}_i$ has at least one different property to the set $\mathcal{T}_{i+1}$ $1 \leq i < k$.

A transformation $\mathcal{T}_j \rightarrow \mathcal{T}_{j+1}$ which modifies the data values to conform to a different schema is referred to as a *data translation*. A transformation which alters the location of the data set is referred to as a *data movement*. To distinguish between the two, a data translation is described using the symbol $\rightarrow$, and a movement of data set $\mathcal{T}_j$ from some computer system

x to another computer system y is described as:

$$\mathcal{T}_j(C_j,\ I_j,\ P_j,\ L_x) \rightsquigarrow \mathcal{T}_{j+1}(C_j,\ I_j,\ P_j,\ L_y).$$

This can be abbreviated to $\mathcal{T}_j \rightsquigarrow \mathcal{T}_{j+1}$.

A data transfer will involve at least one data translation or movement, and may have zero or more additional data translations or movements.

A variety of transfers can be described using the notation presented above, according to the different transformations that occur during the transfer process. One of the simplest transfers is where a data set is transferred to a destination geographical information system with the same conceptual, implementation and physical representation of data values. The transfer consists of only a data movement, and is described as:

$$\mathcal{S}(C_S,\ I_S,\ P_S,\ L_S) \rightsquigarrow \mathcal{D}(C_S,\ I_S,\ P_S,\ L_D).$$

When an organisation changes the computer system while retaining the same geographical information system, or a new version of a geographical information system is released with improved physical representations for data values, existing data values may be translated to conform to a new physical schema. This translation is described as:

$$\mathcal{S}(C_S,\ I_S,\ P_S,\ L_S) \rightsquigarrow \mathcal{T}_0(C_S,\ I_S,\ P_S,\ L_D) \rightarrow \mathcal{D}(C_S,\ I_S,\ P_D,\ L_D).$$

When an organisation changes to a different geographical information system, existing data values are likely to have to conform to different implementation and physical schemas. These translations are described as:

$$\mathcal{S}(C_S,\ I_S,\ P_S) \rightarrow \mathcal{T}_0(C_S,\ I_D,\ P_S) \rightarrow \mathcal{D}(C_S,\ I_D,\ P_D).$$

Transforming the data values to conform to a different implementation schema will also result in the data values conforming to a different physical schema. In a practical sense, modifying the type of some value often corresponds to modifying the physical representation of that value. Therefore, these two translations may occur together, rather than one after the other as shown in the above expression.

When an organisation acquires a new data set, these new values often conform to a conceptual schema that is different to the conceptual schema to which the existing data values conform. Furthermore, this new data set is likely to conform to different implementation and physical schemas as a consequence of conforming to a different conceptual schema. Transfer

of the new data set can therefore be described as:

$$\begin{aligned}\mathcal{S}(C_S,\ I_S,\ P_S,\ L_S) \quad &\rightsquigarrow \mathcal{T}_0(C_S,\ I_S,\ P_S,\ L_D)\\ &\rightarrow \mathcal{T}_1(C_D,\ I_S,\ P_S,\ L_D)\\ &\rightarrow \mathcal{T}_2(C_D,\ I_D,\ P_S,\ L_D) \rightarrow \mathcal{D}(C_D,\ I_D,\ P_D,\ L_D).\end{aligned}$$

To illustrate further the value of the notation summarised in Table 3.1, the problem of information loss is discussed below.

3.2.1 Information loss

Consider the transfer $\mathcal{S} \mapsto \mathcal{D}$, comprising the steps:

$$\mathcal{S} \xrightarrow{*} \mathcal{T}_i(C_i,\ I_i,\ P_i) \xrightarrow{*} \mathcal{T}_j(C_j,\ I_j,\ P_j) \xrightarrow{*} \mathcal{D}.$$

Information may be lost during this transfer as a consequence of either differences between the source and destination schemas, or the transformations that form the transfer.

3.2.1.1 Differences between source and destination schemas

Differences between the source and destination schemas may occur either because of:

incompatible schemas

The conceptual, implementation, and physical schemas C_S, I_S, and P_S of the source geographical information system define a representation for real world phenomena that cannot be represented using the schemas C_D, I_D, and P_D of the destination geographical information system; or

incompatible data models

The concepts provided by the data models for expressing the source schemas C_S, I_S, and P_S, are not provided by the models used for expressing the destination schemas C_D, I_D, and P_D.

In the case of incompatible schemas, the destination schemas could, in rare circumstances, be modified to define the necessary representation for the source data values. In the case of incompatible data models, the problem is more serious since changing the data models of the destination schemas will require choosing some other data model(s) that satisfy the requirements of both the existing data values at the destination, and the data values transferred from the source. Choosing a different data model may even require a different geographical information system.

3.2.1.2 During the transfer process

Loss of information during the transfer process occurs if values representable in both the source and destination data sets are removed during some translation to conform to some temporary schemas. An example of losing data for this reason is when precise representations for a circle are defined by the source and destination schemas, and during the transfer process the data must be translated to conform to schemas that have no defined representation for circles. This transfer is described as: $\mathcal{S} \xrightarrow{*} \mathcal{T}_0 \xrightarrow{*} \mathcal{D}$ where: the data set $\mathcal{S}$ contains circular data values represented according to the conceptual, implementation and physical schemas; $\mathcal{T}_0$ does not have circular data values because the schemas to which this data set conforms do not define a representation for such values; and $\mathcal{D}$ does not have circular data values, even though the destination schemas define a representation for these values, because they had to be removed by the transformations $\mathcal{S} \xrightarrow{*} \mathcal{T}_0$.

This type of information loss can only be avoided if the transfer process is altered so that the data values do not have to conform to such a temporary schema. That is, the transfer becomes $\mathcal{S} \xrightarrow{*} \mathcal{D}$.

3.3 Interfaces

Translation of one set of data values into another is accomplished using an *interface*, 'a mechanism by which one data structure can be directly converted into another data structure for the purpose of communication between systems or sub-systems' (van Roessel *et al* 1986).

When data was moved from one system to another on magnetic tape, the transfer could be described by either:

$$\mathcal{S} \rightsquigarrow \overbrace{\underbrace{\mathcal{T}_1 \xrightarrow{*} \mathcal{T}_k}_{a} \xrightarrow{*} \mathcal{D}}^{\text{an interface}}$$

where the data translations (a) occur *after* the data has been moved to the destination computer system, or

$$\mathcal{S} \overbrace{\underbrace{\rightarrow \mathcal{T}_1 \xrightarrow{*} \mathcal{T}_k}_{b}}^{\text{an interface}} \rightsquigarrow \mathcal{D}$$

where the data translations (b) occur *before* the data has been moved to the destination computer system.

The movement of data from one computer system to another can be a part of an interface by including software to control the movement of data through a communications network.

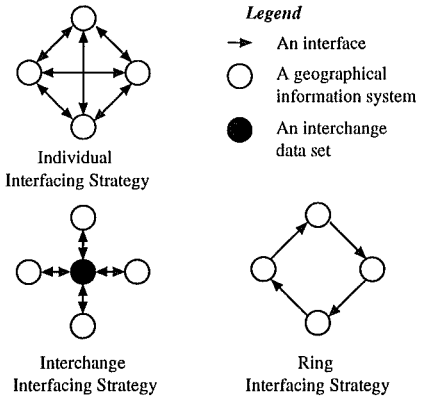


Figure 3.1: Commonly used interfacing strategies

Transfers with data movement occuring in between data translations can be described as:

$$\mathcal{S} \stackrel{*}{\rightarrow} \mathcal{N}_S \rightsquigarrow \mathcal{N}_D \stackrel{*}{\rightarrow} \mathcal{D}.$$

Developing interfaces for performing this type of transfer is to be discussed in Chapter 4.

3.4 Interfacing strategies

When data is to be transferred between a number of geographical information systems, the number of interfaces required to accomplish these transfers is determined by the method adopted for interfacing the geographical information systems. An *interchange group* is defined here to consist of N geographical information systems among which data is to be transferred. An *interfacing strategy* is the method governing the use of interfaces to allow data transfer among members of the interchange group.

Fosnight & van Roessel (1985) have described three interfacing strategies (Figure 3.1):

1. The *individual* interfacing strategy, where a separate interface is constructed for a data transfer $\mathcal{S} \mapsto \mathcal{D}$ between each pair of geographical information systems.

2. The *ring* interfacing strategy, where members of the interchange group are interfaced in series, with the last in the series connected to the first. A data transfer $\mathcal{S} \mapsto \mathcal{D}$ is described as:

$$\mathcal{S} \stackrel{*}{\rightarrow} \mathcal{D}$$

when the destination geographical information system is the next in the series after the source geographical information system, or

$$\mathcal{S} \xrightarrow{*} \mathcal{G}_1 \xrightarrow{*} \cdots \xrightarrow{*} \mathcal{G}_i \xrightarrow{*} \cdots \xrightarrow{*} \mathcal{G}_k \xrightarrow{*} \mathcal{D}$$

where each data set $\mathcal{G}_i$ $1 \leq k \leq N-2$, conforms to the representation required for the i^{th} geographical information system occurring between the source and destination geographical information system.

3. The *interchange* interfacing strategy, where any transfer $\mathcal{S} \mapsto \mathcal{D}$ is accomplished by two interfaces: one, referred to here as the *export interface*, transforms the source data set $\mathcal{S}$ into a data set $\mathcal{I}$ conforming to some interchange format (discussed later), and another, the *import interface*, transforms the data set $\mathcal{I}$ into the destination data set $\mathcal{D}$. The complete transfer is described as:

$$\mathcal{S} \xrightarrow{*} \mathcal{I} \xrightarrow{*} \mathcal{D}.$$

An interchange format is analogous to a geographical information system in that an interchange format provides the concepts and constructs for defining data representations. An important example of an interchange format is the Spatial Data Transfer Standard (SDTS) (Geological Survey 1992).

3.4.1 Comparison

To compare the individual, the ring, and the interchange strategies, consider that:

- there are N geographical information systems in the interchange group, and that N may be large;
- the transfer of data among all geographical information systems must be possible; and
- the time taken to transfer the data from the source to the destination geographical information system is proportional to the number of interfaces required to transfer the data.

The individual, ring, and interchange interfacing strategies are compared on three criteria:

Interfacing effort

the number of interfaces to be constructed if data is to be transferred among all members of an interchange group,

Intervening data representations

the average number of different representations to which the data must conform during the transfer. The data representations considered are those defined for the member geographical information systems and any interchange format, and

Performance

assessed by the average number of interfaces used to transfer data between two members of an interchange group.

In the past, performance has not been an important consideration because accomplishing the data translation was a big enough hurdle. Also, if the data transfer was accomplished by posting data on a magnetic tape, then the translation time was insignificant and 'Simplicity of the transfer, rather than its efficiency, is the key factor' (Penny 1986). Use of communication networks will, however, require much greater attention to performance.

The three strategies are ordered in Table 3.2 from best to worst for each of the criteria described above. This ordering shows that the ring strategy is the worst insofar as the average time taken to perform a data transfer, and the potential for losing information during a data transfer. The ring strategy is, therefore, rarely adopted.

	Comparison criteria		
Ranking	*Interfacing effort*	*Intervening data representations*	*Performance*
Best	ring (N)	individual (0)	individual (1)
Moderate	interchange ($2N$)	interchange (1)	interchange (2)
Worst	individual ($N^2 - N$)	ring ($(N/2) - 1$)	ring ($N/2$)

Table 3.2: Comparing the individual, interchange, and ring interfacing strategies on three criteria

The individual interfacing strategy is ranked as the best in all but the number of interfaces that have to be constructed. Using the individual strategy guarantees that no unnecessary loss of information will occur since the data is not transformed to conform to the schemas of any intervening geographical information systems. Furthermore, this strategy offers the best performance in that only one interface is used to transfer data, regardless of the number of geographical information systems in the interchange group. The dominating drawback of this strategy is the very large number of interfaces, $(N^2 - N)$, that have to be constructed if data is to be transferred among all geographical information systems of the interchange group.

The interchange interfacing strategy provides a reasonable compromise in terms of the number of interfaces to be constructed, the number of interfaces used in any data transfer,

and the potential for losing information during a data transfer. The risk of losing information is low because all data is transformed into only one temporary data set $\mathcal{I}$, conforming to the the interchange format being used.

3.4.2 Interchange interfacing strategy

Although the interchange interfacing strategy is widely adopted, there are some drawbacks to this strategy. One drawback is the additional complexity of interfaces because of the comprehensive nature of an interchange format. Another is the potential for disjoint interface groups as a consequence of many interchange formats being defined.

3.4.2.1 Complexity of interfaces

To avoid any loss of information when an interchange format is involved during the transfer, a comprehensive interchange format must be defined to allow the transfer of all types of data processed by members of an interchange group. Thus, for any data set $\mathcal{A}(C_A,\ I_A,\ P_A)$, there must be an equivalent data set $\mathcal{I}(C_I,\ I_I,\ P_I)$. A comprehensive interchange format must therefore provide all the constructs necessary for expressing the conceptual, implementation and physical schemas C_I, I_I, and P_I of any data set being transferred.

As a consequence of serving a large number of different geographical information systems, a comprehensive interchange format will inevitably be complex. The Spatial Data Transfer Standard (SDTS, FIPS 173) (Geological Survey 1992) is an interchange format that is sufficiently comprehensive to justify the definition of smaller formats, called profiles. Profiles are self-contained subsets of the SDTS. Two profiles are being developed: one for topological vector data, and the other for raster data.

Although the number of interfaces required by the interchange interfacing strategy is far fewer than those required by the individual interfacing strategy, the definition of a comprehensive interchange model requires more complex interfaces to be constructed than those needed for use in the individual interfacing strategy. The additional complexity is a consequence of import and export interfaces having to process all types of data supported by a comprehensive interchange format. If an export interface $\mathcal{S} \xrightarrow{*} \mathcal{I}$ and an import interface $\mathcal{I} \xrightarrow{*} \mathcal{D}$ process only a subset of the data types supported by the interchange format, the two interfaces $\mathcal{S} \xrightarrow{*} \mathcal{I}$ and $\mathcal{I} \xrightarrow{*} \mathcal{D}$ may use different subsets, in which case data cannot be transferred.

Consider an interchange format that provides equivalent vector and tessellation data representations. If the export interface $\mathcal{S} \xrightarrow{*} \mathcal{I}$ places the exported data into the vector representation of the interchange format and the import interface $\mathcal{I} \xrightarrow{*} \mathcal{D}$ imports data from

the tessellation representation of the interchange format, then import and export interfaces cannot be used to transfer data from the source to the destination geographical information systems even though both interfaces use the same interchange format.

3.4.2.2 Disjoint interchange groups

The definition of many different interchange formats limits the effectiveness of the interchange interfacing strategy because an interchange group will be formed for each of these interchange formats. For example, if the transfer $\mathcal{S} \mapsto \mathcal{D}$ is desired, but the source and destination geographical information systems belong to different interchange groups, each based on a different interchange format, and that from previous transfers the interfaces $\mathcal{S} \xrightarrow{*} \mathcal{I}_1$ and $\mathcal{I}_2 \xrightarrow{*} \mathcal{D}$ have been constructed, then to achieve the desired transfer either:

- some public-spirited person would have to construct the interface $\mathcal{I}_1 \xrightarrow{*} \mathcal{I}_2$, which would be a difficult task given the expected complexity of the interchange formats $\mathcal{I}_1$ and $\mathcal{I}_2$, or, more likely,
- one of the two interfaces $\mathcal{S} \xrightarrow{*} \mathcal{I}_2$ or $\mathcal{I}_1 \xrightarrow{*} \mathcal{D}$ would have to be constructed.

Either way, the advantages of using the interchange interfacing strategy have been reduced by the definition of more than one interchange format.

An obvious possibility is to reduce the number of different interchange formats with the aim of eventually having a single, universally accepted format, U. This may occur over time as vendors of geographic information systems start to interface their own system with those interchange formats that become widely used. Reducing the variety of interchange formats would be accelerated if providing an interface to a particular interchange format became a selling point.

There are, however, potential problems in having one universal interchange format. Suppose that a 'universal' standard format U became widely accepted as the interchange format for a large number of geographical information systems. Experience in using the standard may reveal desirable extensions or improvements, leading to the replacement of the standard by a revised version U'.

Adopting U' would mean (again theoretically) all export and import interfaces must be replaced simultaneously to maintain a complete set of interfaces. Since that is likely to be impractical, a situation can be foreseen where different export and import interfaces must be kept on hand for each version of the standard. Alternatively, one might try to construct each interface in a way which allows for different versions of the standard. Whatever is done, a fair amount of confusion seems inevitable.

3.5 Conclusions

The interchange interfacing strategy is a reasonable method for governing the use of interfaces to transfer data among several geographical information systems. This strategy offers a reasonable compromise between the number of interfaces to be constructed, and the potential for losing information during a data transfer. Adopting the interchange interfacing strategy does, however, require: the definition of a comprehensive interchange format, a complicated problem in itself; and the construction of complex interfaces to use this format.

The individual interfacing strategy is the ideal method for governing the use of interfaces because data from the source geographical information system is transferred directly into the representation required for the destination geographical information system. Any information lost during the transfer should be a consequence of only the differences between the source and destination representations, and therefore unavoidable. Adopting the individual interfacing strategy is impractical, however, because of the large number of interfaces that have to be constructed.

Whichever interfacing strategy is used, methods of interface construction are needed that are simple and systematic, and result in interfaces that are easy to modify. The primary objective pursued in this thesis is to develop such a method of interface construction.

Given the increasing use of communication networks for transferring data, these methods should allow the construction of interfaces that transfer data through communication networks. Communication networks and their role in data transfers are discussed further in the next Chapter.

Interfaces and communication networks

Chapter 4

Communication networks are increasingly being used for moving data among computer systems, as described earlier in Section 3.1. A *communicating interface*, to be discussed in Section 4.1, is an interface that performs data translations and sends data to or receives data from another interface through a communications network.

In the past, the function of an interface has been typically restricted to translating data from one file format into another file format. Use of communication networks by interfaces will result in widening the functions of interfaces. In Section 4.2 a discussion is presented on how communicating interfaces may provide additional services, including: various methods of requesting the data to be transferred; and access to a data directory whose entries describe data sets that are accessible through the communication network. Ultimately, communicating interfaces are likely to be incorporated in geographical information systems to allow the direct transfer of data among different geographical information systems.

Any interface that communicates with another must use a common network representation. In Section 4.3, an analogy is drawn between this common network representation and the interchange format which forms the basis of the interchange interfacing strategy. Interchange formats and the interchange interfacing strategy were described in Section 3.4.

Increasingly, ISO-OSI standards are being used in the transfer of geographical data. Use of ISO-OSI standards by interchange formats such as the SDTS (Geological Survey 1992) and MACDIF (Evangalatos & Allam 1991) for representing data are discussed in Sections 4.4.1 and 4.4.2 respectively. Section 4.4.3 is a description of the proposed Geographic Document Architecture (GDA) (McKellar, O'Brien & Lalonde 1990). According to this architecture, data is transferred using a '*vessel*', which may carry '*containers*' of geographical data across

different communication '*channels*'.

In this thesis, communicating interfaces are constructed within the ISO Development Environment (Rose *et al* 1991) according to a general structure for applications which communicate with each other using a communications network. In Section 4.5, the general structure of these applications is described together with a brief introduction to software tools `rosy` and `pepsy` which are provided within ISODE for assisting the construction of these applications.

4.1 Communicating interfaces

Transferring data through communication networks allows translations to be performed on several different computer systems. Therefore, a transfer $\mathcal{S} \mapsto \mathcal{D}$ may have the following general form:

$$\underbrace{\mathcal{S} \xrightarrow{*} \mathcal{T}_i}_{L_S} \rightsquigarrow \underbrace{\mathcal{T}_j \xrightarrow{*} \mathcal{T}_k}_{L_T} \cdots \underbrace{\mathcal{T}_l \xrightarrow{*} \mathcal{D}}_{L_D}$$

where the transformations $\mathcal{S} \xrightarrow{*} \mathcal{T}_i$ are performed by a software module operating on computer system L_S, the transformations $\mathcal{T}_j \xrightarrow{*} \mathcal{T}_k$ are performed by another software module operating on computer system L_T, and so on.

Van Roessel *et al* (1986) define an interface as 'a mechanism by which one data structure can be directly converted into another for the purpose of communication between systems or sub-systems'. Each software module may be regarded as an interface according to that definition, and the transfer $\mathcal{S} \mapsto \mathcal{D}$ is therefore performed by a collection of interfaces. These software modules may, however, communicate with each through a communications network, as well as transform data values.

A *communicating interface* is defined as a software package that transforms data from one representation to another and may either receive the data to be transformed from another communicating interface, send the transformed data on to another communicating interface, or both send and receive data with another communicating interface.

Any pair of communicating interfaces must have a common representation for data values sent through the communications network. When more than two communicating interfaces have a common network representation, this representation has a role similar to that of an interchange format discussed earlier in Section 3.4. These communicating interfaces link different geographical information systems to form an interchange group in which transfers are performed according to the interchange interfacing strategy. Communicating interfaces and interchange formats are discussed further in Section 4.3.

A practical example of communicating interfaces is discussed in Chapter 12. This example comprises several communicating interfaces which have been constructed by the author. Each of these interfaces transforms data either into or from a common representation for transmission through an Ethernet communications network. A *source communicating interface* transforms data values into a data set $\mathcal{N}_S$, conforming to the common network representation and located on the source computer system L_S. Examples of the transformations performed by these source communicating interfaces are:

$\mathcal{S}_{cmap} \xrightarrow{*} \mathcal{N}_S$

where $\mathcal{S}_{cmap}$ is a data set that conforms to the Colourmap data file format (CSIRONET 1986) and is located on the source computer system; and

$\mathcal{S}_{gina} \xrightarrow{*} \mathcal{N}_S$

where $\mathcal{S}_{gina}$ is a data set that conforms to the Gina data file format (GeoVision 1986) and is located on the source computer system.

A *destination communicating interface* transforms data values $\mathcal{N}_D$ into a variety of other data sets. The data set $\mathcal{N}_D$ conforms to the common network representation and is located on the destination computer system L_D. Examples of the transformations performed by these interfaces are:

$\mathcal{N}_D \xrightarrow{*} \mathcal{D}_{gina}$

where $\mathcal{D}_{gina}$ is a data set that conforms to the Gina data file format and is located on the destination computer system; and

$\mathcal{N}_D \xrightarrow{*} \mathcal{D}_{POSTSCRIPT}$

where $\mathcal{D}_{POSTSCRIPT}$ is a data set comprising values represented using constructs of the PostScript (Adobe 1985) language and located on the destination data system.

Since the source and destination communicating interfaces have a common network representation, they may be combined to accomplish a variety of data transfers. Examples of these transfers are:

$\mathcal{S}_{gina} \xrightarrow{*} \mathcal{N}_S \rightsquigarrow \mathcal{N}_D \xrightarrow{*} \mathcal{D}_{gina}$

where a data set $\mathcal{S}_{gina}$ conforming to the Gina data file format on the source computer system, is transferred into a data set $\mathcal{D}_{gina}$ also conforming to the Gina data file format, but located on the destination computer system;

$\mathcal{S}_{cmap} \xrightarrow{*} \mathcal{N}_S \rightsquigarrow \mathcal{N}_D \xrightarrow{*} \mathcal{D}_{gina}$

where a data set $\mathcal{S}_{cmap}$ is transferred into a data set $\mathcal{D}_{gina}$;

$\mathcal{S}_{gina} \xrightarrow{*} \mathcal{N}_S \rightsquigarrow \mathcal{N}_D \xrightarrow{*} \mathcal{T}_G \xrightarrow{*} \mathcal{D}_{POSTSCRIPT}$

where a data set $\mathcal{S}_{gina}$ is transferred into a data set $\mathcal{D}_{POSTSCRIPT}$; and

$\mathcal{S}_{cmap} \xrightarrow{*} \mathcal{N}_S \rightsquigarrow \mathcal{N}_D \xrightarrow{*} \mathcal{T}_G \xrightarrow{*} \mathcal{D}_{POSTSCRIPT}$

where a data set $\mathcal{S}_{cmap}$ is transferred into a data set $\mathcal{D}_{POSTSCRIPT}$.

Use of these and other communicating interfaces will be described further in Chapter 12. In constructing these communicating interfaces, the author realized that additional functions indirectly related to geographical data transfer could have been provided by these communicating interfaces.

4.2 The functions of communicating interfaces

Allowing interfaces to control the movement of data among computer systems widens the kinds of functions an interface can perform. As well as translating data among different representations, interfaces may provide other functions such as a directory service for locating data sets accessible through the network, and a query service that provides various methods for specifying what data is to be transferred.

The system to be described in Chapter 12 comprises an 'ad hoc' directory service that is shared by all the source communicating interfaces, but is inaccessible to any destination communicating interface. Consequently, users must have prior knowledge of the entries in this directory. An entry comprises the name of a data file and the spatial domain of the data in the file. Source communicating interfaces use this directory to select data when responding to transfer queries from destination communicating interfaces. Future implementations of source and destination communicating interfaces should allow users of a destination communicating interface access to the directory of the source communicating interface.

A more general approach is to use the X.500 directory (Rose *et al* 1991, Vol 5). This directory might be used for providing descriptions of available data sets in the same way that this directory is used for peoples' names, addresses, and even photographs. Source communicating interfaces should maintain entries in the X.500 directory which describe accessible data sets. Destination communicating interfaces should provide users with facilities to search the directory for entries which describe the data sets that can be transferred and the source communicating interfaces to transfer these data sets.

Communicating interfaces could also provide additional methods for requesting data to be transferred. Data to be transferred by the interfaces described earlier in Section 4.1, for example, are requested by specifying either the name of the file in which the data is represented, or the bottom left and top right corners of the spatial domain of the data set.

In general, any group of communicating interfaces will have to provide a set of 'core operations' to allow each interface to communicate with others. A definition of these core operations, along with a definition of a network representation, may eventually form a standard interchange format to which communicating interfaces must conform.

4.3 Communicating interfaces and interchange formats

Some interchange formats define a file-based physical representation for data values. The SDTS, for example, adopts the file representation defined by ISO 8211 (ISO8211 1985), to be described in Section 4.4.1. Others, such as MACDIF, adopt a network representation defined by a combination of ISO 8824 and 2375 (ISO8824 1987, ISO2375 1985), as will be explained in Section 4.4.2. To combine these different approaches to interchange formats, a mechanism is needed such as the Geographic Data Architecture proposed by McKellar, O'Brien & Lalonde.

The key idea behind the Geographic Data Architecture, to be discussed in Section 4.4.3, is to structure interchange formats in such a way that allows a variety of physical schemas to be defined for any one interchange format. Each schema defines a physical representation suitable for a different communication medium. A variety of data sets $\mathcal{I}_i$, $i = 1 \ldots n$ may be created that conform to the same conceptual and implementation schemas C_I and I_I, but conform to different physical schemas P_i, $i = 1 \ldots n$. Thus, the sets $\mathcal{I}_{file}(C_I,\ I_I,\ P_{file})$, and $\mathcal{I}_{network}(C_I,\ I_I,\ P_{network})$ conform to the same conceptual and implementation schemas C_I and P_I, but to different physical schemas P_{file} and $P_{network}$ which are appropriate for different communication media.

Development of interchange formats is increasingly drawing on the use of ISO-OSI standards, which are designed for use according to the ISO-OSI reference model, described below.

4.4 The ISO-OSI Reference Model

Computer systems communicate with each other across a network using a collection of rules and conventions known as *communication protocols*. These protocols are partitioned into layers, with a layer n protocol designed to use the services provided by a layer $n-1$ protocol.

In theory, two computer systems communicate at all layers of the network architecture. At each layer, peer processes on each system communicate using the protocol defined at that layer. In practice, the layer n processes on the source and destination computer system communicate by passing data to the layer $n-1$ processes on each system. At the lowest layer,

the source process transmits data through the communication network to the corresponding process on the destination system.

As data values are passed from the highest to the lowest layer, the data values are transformed from the representation required by an application into the representation required for transmission through some communication medium such as a telephone line or a satellite channel. At the highest layer, an application may transfer a person's name to another application on a different computer system. After passing through to the lowest layer, the person's name, along with additional information introduced by intermediate layers such as the size, type, and the address of the destination computer system, is represented as a sequence of bits for transmission along the communication medium.

The layered structure of computer networks requires a precise definition of the services provided by each layer n in the architecture, and how these services are accessed by the next layer $n+1$. Given such a precise definition, different implementations of layer n may be used by layer $n+1$, provided that all the different implementations make available the defined services which are accessed in the specified manner for the layer.

The International Standards Organisation (ISO) has defined a seven layered network architecture called the Reference Model of Open Systems Interconnection (OSI) (ISO7498 1984). A wide variety of protocols have been designed for this model, some of which have been used in geographical data transfers as will be described in Sections 4.4.1 to 4.4.3. The ISO-OSI reference model consists of seven layers:

the *application* layer

governs the services provided by the network which are accessible to an application, and any data transformations necessary for the data values to conform to the types of values transmitted across the network;

the *presentation* layer

governs the network representation of the data being transmitted, and the conversion of data between local and network representations;

the *session* and *transport* layers

govern the connection between the source and destination computer systems, and maintain the integrity of the data during transmission; and

the *network*, *data link*, and *physical* layers

govern the transmission of data across the physical media such as wires, microwave links, and satellite channels.

A detailed discussion of communication protocols at each of these levels is beyond the scope of this thesis. However, it is important to appreciate that any pair of communicating interfaces will use the same protocols at these layers. Any pair of communicating interfaces which transmit geographical data through a communication network use protocols defined at the different layers of the ISO-OSI reference model.

A pair of communicating interfaces performs a data transfer $\mathcal{S} \xrightarrow{*} \mathcal{N}_S \rightsquigarrow \mathcal{N}_D \xrightarrow{*} \mathcal{D}$. This transfer consists of one or more transformations that may be divided into three groups: application layer transformations, presentation layer transformations, and transformations that occur at the remaining layers of the architecture. A transfer may therefore be described as:

$$\begin{array}{lccc} \text{Application layer} & \mathcal{S} \xrightarrow{*} \mathcal{T}_i & & \mathcal{T}_j \xrightarrow{*} \mathcal{D} \\ \text{Presentation layer} & \mathcal{T}_i \xrightarrow{*} \mathcal{N}_S & & \mathcal{N}_D \xrightarrow{*} \mathcal{T}_j \\ \text{Layers 1–5} & & \mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D & \end{array} \tag{4.1}$$

This transfer includes two sequences of application layer transformations, and two sequences of presentation layer transformations. Although important, transformations occurring at the other 5 layers are not considered any further in this thesis.

The two sequences of application layer transformations are:

$\mathcal{S} \xrightarrow{*} \mathcal{T}_i$

which transform the original data set $\mathcal{S}$, conforming to the conceptual and implementation schemas of the source application, into a temporary data set $\mathcal{T}_i$ conforming to the conceptual and implementation schemas of the network representation common to both communicating interfaces; and

$\mathcal{T}_j \xrightarrow{*} \mathcal{D}$

which transform the temporary data set $\mathcal{T}_j$, conforming to the conceptual and implementation schemas of the network representation common to both communicating interfaces, into the destination data set $\mathcal{D}$, conforming to the conceptual and implementation schemas specific to the destination application.

The temporary sets $\mathcal{T}_i$ and $\mathcal{T}_j$ comprise values that conform to the same conceptual and implementation schemas, but may conform to different physical schemas as a consequence of the two computer systems having different hardware.

The two sequences of presentation layer transformations are:

$\mathcal{T}_i \xrightarrow{*} \mathcal{N}_S$

which transform the data set $\mathcal{T}_i$ into the data set $\mathcal{N}_S$ conforming to the conceptual,

implementation, and physical schemas of the network representation common to both the source and destination applications; and

$\mathcal{N}_D \stackrel{*}{\rightarrow} \mathcal{T}_j$

which transform the data set $\mathcal{N}_D$, conforming to the conceptual, implementation, and physical schema of the network representation common to both applications, into the data set $\mathcal{T}_j$, conforming to the local physical schema of the representation used by the destination application.

All the data sets $\mathcal{T}_i$, $\mathcal{N}_S$, $\mathcal{N}_D$, and $\mathcal{T}_j$ of the presentation layer conform to the same conceptual and implementation schemas and the data sets $\mathcal{N}_S$ and $\mathcal{N}_D$ conform to the same physical schema.

Application and presentation layer communication protocols developed by the ISO and the International Telegraph and Telephone Consultative Committee (CCITT) are increasingly being used in transfers of geographical data. For example, the standard data descriptive file for information interchange (ISO8211 1985) is used in the SDTS, as will be described in Section 4.4.1, and the MACDIF interchange format is defined using Abstract Notation.1 (ASN.1) (ISO8824 1987), as will be described in Section 4.4.2. A Geographic Document Architecture (GDA) (McKellar *et al* 1990), to be described in Section 4.4.3, is also being developed along similar lines to the existing Office Document Architecture (ODA) standard (ISO8613-1 1989).

4.4.1 ISO 8211 and the Spatial Data Transfer Standard

A substantial part of the Spatial Data Transfer Standard (SDTS) (FIPS 173), is the logical specification of spatial data transfer modules, consisting of records, fields and subfields. These logical specifications are implemented using the file representation defined by ISO 8211, which is described in Appendix A.1. Spatial data transfer modules are designed to allow the transfer of information such as data quality, feature and attribute data dictionary, coordinate reference, spatial object, and associated attribute and graphic symbology information (Fegeas, Cascio & Lazar 1992).

To illustrate the use of ISO 8211 in the SDTS, the logical specification of a Catalog/Directory module (Geological Survey 1992, Table 12, pg 65) is given in Table 4.1, and an implementation of this module using ISO 8211 is given in Figure 4.1. The Catalog/Directory module is used to specify the text files in which the various SDTS modules are being transferred.

The SDTS is designed primarily for data transfers in which either magnetic tape or CD-ROM is used for moving data to different computer systems. MACDIF is an alternative

FIELD NAME	SUBFIELD NAME	FIELD/SUBFIELD DESCRIPTION	TYPE	DOMAIN	DOMAIN DESCRIPTION	MNEMONIC
Catalog/ Directory(P)[M]						CATD
	Module Name[M]	A unique identifier for this Catalog/Directory module.	A	Alphanum	Name shall begin with alphabetic character other than SPACE.	MODN
	Record ID[M]	A number for the module record, unique within the module.	I	Integer	Unsigned integer; with Module Name shall form unique ID within the file set.	RCID
	Name[M]	The unique value in the Module Name subfield of the module referenced.	A	Alphanum	Name shall begin with alphabetic character	NAME
	Type[M]	The transfer module primary field name of "Name" above.	A	Gr-chars	Shall be a valid module primary module field name.	TYPE
	Volume	The volume on which a part or all of the module is to be found.	A	Gr-chars	Valid volume descriptor, wild card characters may be used.	VOLM
	File	The file on which a part or all of the module is to be found.	A	Gr-chars	Valid file name, wild card characters may be used.	FILE
	Record	The record on which a part or all of the module is to be found (implementation record).	I—A	Integer Gr-chars	Unsigned integer, wild card characters may be used.	RECD
	Comment	Any other information.	A	Gr-chars	Any combination of graphics characters.	COMT

Table 4.1: Logical specification of a Catalog/Directory transfer module of the SDTS (Geological Survey 1992, Table 12,pg65)

```
002103L   0600064   4504000000470000000001003000047CATD006900077^^
0000;&DLG-Optional/SDTS Transfer File^_0001CATD^^
0100;&DDF RECORD IDENTIFIER^_^_^^
2600;&CATALOG/DIRECTORY^_*MODN!NAME!TYPE!VOLM!FILE!RECD!COMT^_(5A,I,A)^^
01289 D     00298   450400010006000000CATD004500006.....CATD004600945^^
1^^
CD^_CD^_Catalog/Directory^_^_catalog.ddf^_1^_10/88^^
CD^_LG^_Lineage^_^_quality.ddf^_1^_10/88^^
CD^_PA^_Positional Accuracy^_^_quality.ddf^_2^_10/88^^
CD^_AA^_Attribute Accuracy^_^_quality.ddf^_3^_10/88^^
....
CD^_L1^_Line^_^_dlg_sdts.ddf^_41^_10/88 - DLG Lines^^
```

Figure 4.1: An example of an SDTS Catalog/Directory Module according to specification in Table 4.1

interchange format defined by the Canadians for transferring data through communications networks.

4.4.2 ISO 8824/8825 and MACDIF

The Map And Chart Data Interchange Format (MACDIF) (Evangalatos *et al* 1989, Evangalatos & Allam 1991) is a Canadian standard developed for communicating a wide variety of geographical data through public telecommunication networks. MACDIF may be used to communicate 'anything from raw digitised map information to a fully symbolised and cartographically enhanced map or chart' (Evangalatos *et al* 1989). Furthermore, MACDIF may be used to communicate information for updating existing data sets.

Design of MACDIF has emphasized the use of the ISO-OSI reference model and some of the associated communication protocols. At the application layer, MACDIF describes 'which information may be used in the description of a map or chart and how that information is interrelated' (Evangalatos *et al* 1989). At the presentation layer, MACDIF describes how 'information would be represented in terms of a stream of digital data'.

MACDIF has been described as a twenty-five page abstract syntax using Abstract Syntax Notation.1 (ASN.1) (ISO8824 1987). ASN.1 is summarised in Appendix A.2. A part of the abstract syntax for MACDIF given by Evangalatos *et al* (1989) is reproduced in Figure 4.2. In general, 'MACDIF organizes information into a number of categories which define the overall structure of the spatial data, its relation to a world coordinate system, the features which make up the data set, their attributes and boundaries, and optionally any related symbolization and topological relationships' (*op cit*).

Values of the data types defined using ASN.1 are typically represented according to ISO 8825 (ISO8825 1987), the specification for basic encoding rules for ASN.1. ISO 8825 is

```
Digital-Map-or-Chart ::= SEQUENCE { Map-Header, Map-Def-Section}

Map-Def-Section ::= SEQUENCE OF Map-Definition

Map-Definition  ::= SEQUENCE {
                        [1] Map-Sub-Header OPTIONAL,
                        [2] Transform-Definition-Section,
                        [3] Feature-Definition-Section,
                        [4] Segment-Definition-Section,
                        [5] Topological-Definition-Section OPTIONAL,
                        [6] Symbolization-Definition-Section OPTIONAL,
                        [7] Associated-Information-Definition_Section OPTIONAL }

Map-Header      ::= SEQUENCE {
                        [1] Data-Set-ID,
                        [2] Content-ID,
                        [3] Producer-ID,
                        [4] Reference-Information,
                        [5] Quality-Declaration-section OPTIONAL,
                        [6] Source-Declaration-Section OPTIONAL,
                        [7] History-Description OPTIONAL,
                        [8] Parameter-Definition OPTIONAL }
```

Figure 4.2: Part of an ASN.1 description of MACDIF (Evangalatos *et al* 1989)

summarised in Appendix A.3. However, designers of MACDIF adopted a character-coded approach based upon the use of the standards ISO 2022 (ISO2022 1986), which describes the operation of a code table-oriented data representation, and ISO 2375 (ISO2375 1985), which defines the procedures for registering code tables. Pictorial information is represented by the use of a code table that 'contains thirty-two primitives, such as commands to draw a point, a line, or a polygon, etc' (Evangalatos *et al* 1989). An ISO working group is 'establishing a standard approach to defining code tables for pictorial and numeric information' (*op cit*).

4.4.3 The proposed Geographical Document Architecture

The Geographic Document Architecture (GDA) proposed by McKellar *et al* (1990) is a method of structuring geographical documents for interchange analogous to the method of structuring office documents for interchange specified by the Office Document Architecture (ODA) (ISO8613-1 1989). The underlying principle of GDA is to separate the information content of data at the application layer from the physical representation of data values at the presentation layer.

The GDA is described as a '*vessel*' in which '*containers*' of geographical data are communicated over '*channels*'. The GDA may provide a mechanism for 'unifying the underlying

interchange structure' (McKellar *et al* 1990) since data values within a container are expected to conform to an interchange format such as the SDTS or the MACDIF formats, and are represented according to ISO-OSI standards appropriate for communication over a chosen channel. A standard, such as ISO 8211, may specify an appropriate representation for containers sent over a channel that uses media such as magnetic tape or CD-ROM. Standards such as ISO 8824/8825 may be used for transmission over a channel that uses a telecommunications network.

According to the terminology described in Chapter 2, the GDA is distinguishing between the implementation and physical levels of data abstraction. Data values within an SDTS container, and represented according to the concepts defined by the ISO 8211 specification, for example, correspond to a data set $\mathcal{T}(C,\ I_{SDTS},\ P_{ISO8211})$ where the data set $\mathcal{T}$ conforms to a conceptual schema C; an implementation schema I_{SDTS}, expressed using the model of spatial data defined by the SDTS, and a physical schema $P_{ISO8211}$, expressed according to the concepts defined by ISO 8211.

Given the wide use of ISO-OSI standards, the author has explored the construction of communicating interfaces that use ISO-OSI communication protocols. In particular, communicating interfaces have been constructed by the author using software tools and implementations of ISO-OSI communication protocols provided by the ISO Development Environment (ISODE).

4.5 The ISO Development Environment

The ISO Development Environment (ISODE) (Rose *et al* 1991) was 'developed as a research tool and represents an effort to promote the use of the International Organisation for Standardisation (ISO) interpretation of Open Systems Interconnection (OSI)'. The construction of applications that use ISO-OSI communication protocols is assisted within ISODE by the provision of: an implementation of some network communication protocols defined by ISO, IEC[1], CCITT[2], and ECMA[3]; and some software tools for generating frequently occurring components of these applications from high-level specifications.

An application developed within ISODE may be thought of as a collection of *application service elements*, each performing a function required for the application. Two examples of application service elements are:

[1]International Electrotechnical Commission
[2]International Telegraph and Telephone Consultative Committee
[3]European Computer Manufacturer Association

Application Control Service Element (ACSE)

defined by the standards ISO 8649 and ISO 8650. An ACSE is concerned with the 'task of "starting" and "stopping" the network for the application' (Rose *et al* 1991, Volume 1, page 15); and

Remote Operation Service Element (ROSE)

defined by the standards ISO 9072 (ISO9072-1 1988, ISO9072-2 1988). A ROSE is concerned with the task of requesting and executing remote operations, and dealing with either the results or errors produced by these operations.

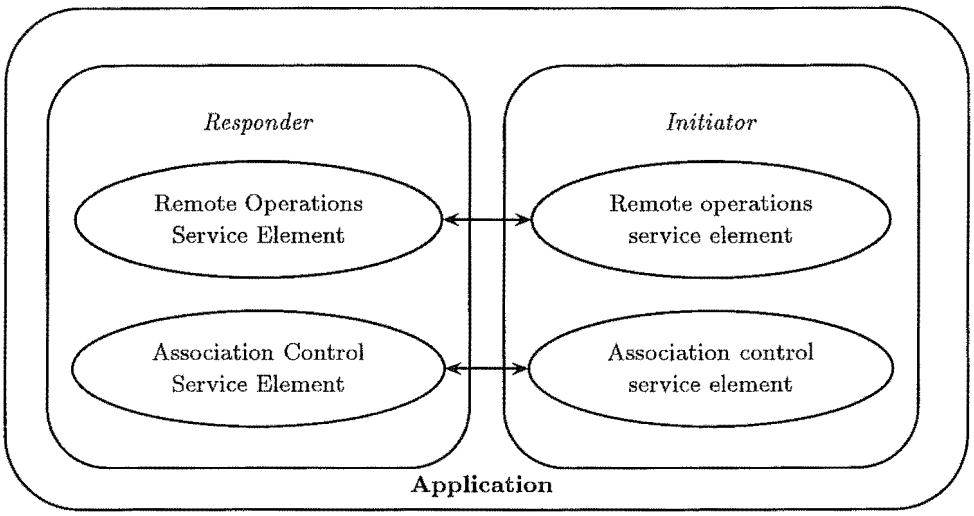


Figure 4.3: Structure of applications

As shown in Figure 4.3, an application may be divided into an initiator and a responder. An *initiator* begins an association with a responder and request operations to be performed by a responder. An initiator may be either interactive or embedded. An *interactive initiator* is such that 'the user runs a program and interactively directs the invocation of operations'. An *embedded initiator* is such that 'as part of its running, the program automatically forms an association and invokes operations as required'. A *responder* accepts an association with an initiator and performs any operations requested by this initiator.

Typically, an initiator and a responder are executed on different computer systems, and an ACSE is used to establish an association between these two parts of an application. This association serves as the basis for further communication between an initiator and a responder which is controlled by a ROSE. The general form of this communication is shown in Figure 4.4.

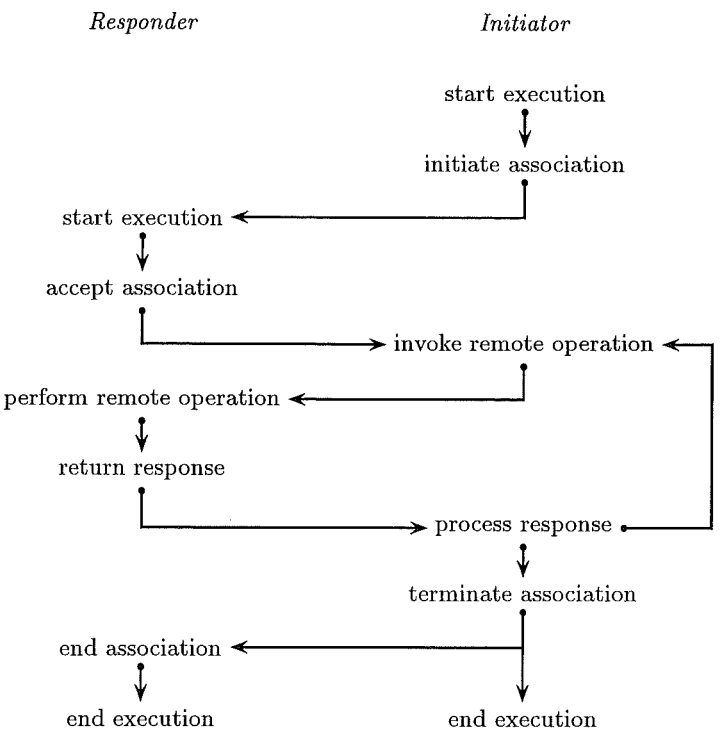


Figure 4.4: Communication between responders and initiators

Consider now a pair of communicating interfaces which perform the following transfer:

$$\mathcal{S} \xrightarrow{*} \mathcal{N}_S \rightsquigarrow \mathcal{N}_D \xrightarrow{*} \mathcal{D}. \tag{4.2}$$

The source communicating interface corresponds to a responder, with an operation performed by the responder being the transformations $\mathcal{S} \xrightarrow{*} \mathcal{N}_S$. The destination communicating interface corresponds to an initiator, with the sequence of data transformations $\mathcal{N}_D \xrightarrow{*} \mathcal{D}$ being the processing of the response from the responder. In Chapter 11, a detailed explanation is given of the methods used by the author to incorporate source and destination communicating interfaces within applications constructed with the assistance of ISODE.

Any application has a set of abstract data types which define the different kinds of data values sent between a responder and an initiator. These values correspond to the data sets $\mathcal{N}_S$ and $\mathcal{N}_D$ shown above in (4.2). As was explained earlier in Section 2.3, a collection of abstract data types is defined by a pair of syntaxes. An abstract syntax defines a collection of data types without specifying the physical representation for values of these types. The physical representation of these types of values is specified by a concrete syntax. The concrete syntax

for any application constructed within ISODE is defined by the communication standard ISO 8825 (ISO8825 1987).

The abstract syntax defining a collection of abstract data types is specified within a *remote operations module*, together with the operations and associated error codes that are to be managed by the ROSE of the application. A remote operations module is processed by the software tool `rosy`, which generates a collection of macros and C function declarations for interacting with the run-time library provided by ISODE.

The abstract syntax contained within a remote operations module is passed on by the software tool `rosy` for processing by another software tool called `pepsy`. The software tool `pepsy` processes an abstract syntax and generates a collection of C data structure declarations which define a memory-resident representation for these types of data values, and a set of tables for transforming values between this memory-resident representation, and *presentation elements*, objects 'used to represent a data structure in a machine-independent form' (Rose *et al* 1991, Vol. 1, page 124). These data transformations correspond to the presentation layer transformations $\mathcal{T}_S \xrightarrow{*} \mathcal{N}_S$ and $\mathcal{N}_D \xrightarrow{*} \mathcal{T}_D$ shown in (4.1), the general description given in Section 4.4 of data transfers performed by communicating interfaces.

More detailed explanations of the software tools `pepsy` and `rosy` are given in Chapter 11, together with a more detailed discussion of constructing pairs of source and destination communicating interfaces in the form of applications. However, before discussing the approach to constructing interfaces suggested in this thesis, a review is presented in the next Chapter of various approaches that have been adopted for constructing interfaces that transfer a wide variety of data. This review includes approaches taken in fields other than geographical information systems.

A review of interface construction

Chapter 5

This Chapter is a review of different approaches to interface construction. Methods for constructing interfaces that will transfer data (of many kinds) among different representations are reviewed, together with the construction of interfaces specifically for transferring geographical data. Approaches developed for constructing interfaces in areas other than geographical information systems should be examined to see whether they may be adapted for constructing geographical interfaces. In particular, Section 5.1 contains a review of:

EXPRESS
: a system designed for generating software that transforms hierarchical databases;

SNAP
: an application generator that uses Set Notation as A Programming Language for specifying the desired data transformations;

the Chameleon Project
: a system designed for generating software that transforms electronic manuscripts between different representations; and

the Nagiya Project
: a continuation of the Chameleon Project to investigate data translation in general.

Research into constructing geographical interfaces is reviewed in Section 5.2. Included in this review are:

- the descriptions of two systems:

GEOLINK

GEOLINK (Waugh & Healy 1986) is a general purpose interfacing tool for transforming data produced by one system such as the Oracle database system into a form that can be processed by another such as the GIMMS geographical information system;

SDRSS

SDRSS (Spatial Data Research and Support System) (van Roessel *et al* 1986) is a system comprising a collection of interfaces for translating data among various representations used by geographical information systems including AGIS/GRAM, AMS, ARC/INFO, and INTERGRAPH.

- a discussion of applying the concepts defined by the relational data model to the translation of geographical data; and
- a description of earlier research by the author into the construction of geographical interfaces.

The approach suggested by the author will be presented in Chapter 6.

5.1 General interfaces

In this Section descriptions are given of approaches to constructing interfaces for translating databases, electronic manuscripts, and data in general, between different representations.

5.1.1 EXPRESS

The Extraction, Processing, and REStructuring System (EXPRESS)(Shu *et al* (1977)) is designed for generating interfaces that transform hierarchical databases. The system was 'originally developed as a research prototype in order to test the generality and applicability of applying high-level data description and high-level data manipulation techniques to the data translation problem' (Taylor 1982).

In EXPRESS the translation of hierarchically structured data from a source text file representation into a destination text file representation is specified using two non-procedural languages: DEFINE, for specifying the representation of hierarchically structured data in the source and destination files; and CONVERT, for specifying the translation of data from the source to the destination representation.

A DEFINE specification consists of nested group definitions. Within each group definition, a sequence of data items is described using COBOL-like picture statements. Figure 5.1

is an example of a file specification expressed using DEFINE. The specified file contains, for some cadastral application, a collection of land parcel records, in which are given the address and spatial location of some legal unit of land.

```
FILE PARCEL-DESCRIPTIONS:
     OCCURS FROM 1 TIMES,
     FOLLOWED BY EOF;

  GROUP PARCEL:
    APPEL:    CHAR(4);;

    GROUP ADDRESS:
      OCCURS 1 TIMES;
      HOUSE#:   CHAR(PICTURE IS '999');;
      STREET:   CHAR(40);;
      CITY:     CHAR(30);;
    END ADDRESS;

    GROUP BOUNDARY:
      OCCURS 1 TIMES;
      NLINES:  BINARY(15);

      GROUP LINES:
        OCCURS NLINES TIMES;
        LINEID:  CHAR(6};;
        NPOINTS: BINARY(15);;

        GROUP POINTS:
          OCCURS NPOINTS TIMES,
                 PRECEDED BY '(',
                 INFIXED BY ',',
                 FOLLOWED BY ')';
          X:     CHAR(PICTURE IS '9(3:5)');;
          Y:     CHAR(PICTURE IS '9(3:5)');;
        END POINTS;
      END LINES;
    END BOUNDARY;
  END PARCEL;
```

Figure 5.1: An example of a file specification using the DEFINE language

Repetition of a group is indicated by the OCCURS clause, examples of which are shown in Figure 5.1. The content of a group may be conditional upon the value of a field as shown in Figure 5.2, where the spatial object group consists of either a polygon group or a line group depending upon the value of the OBJ-TYPE field.

The CONVERT language provides nine types of operations for manipulating data represented in hierarchical structures called forms. A *form* may be viewed as a table consisting of one or more fields, each field having one or more subfields. The form shown in Figure 5.3(a),

```
GROUP SPATIAL-OBJECT:
  OCCURS FROM 1 TIMES;
  OBJ-TYPE: CHAR(1);;

  GROUPCASE OB:
    GROUP POLYGON(OBJ-TYPE = 'P'):
       :
       :
    END POLYGON

    GROUP LINE(OBJ-TYPE = 'L'):
       :
       :
    END LINE
```

Figure 5.2: The general DEFINE construction for varying the contents of a group

for example, comprises a collection of parcels, or units of land. Each parcel is uniquely identified by its appellation, typically comprising a Deposited Plan number such as DP4753.

To illustrate the CONVERT language, three of the nine operators will be briefly described. The SLICE operation transforms data represented as a form into data represented as a flat file. For example, the following operation:

```
TEMP1 = SLICE(Appellation, Lines, Points FROM Parcel);
```

when applied to the form shown in Figure 5.3(a) will produce the result shown in Figure 5.4(a).

The SELECT operation creates a new form containing data taken from another form. Data in the new form may have to satisfy a specified selection criteria, expressed as a collection of field comparisons connected together using operators such as AND and OR. For example, the following operation:

```
TEMP2 = SELECT(Appellation, Address(House,Street,City) FROM Parcel
                WHERE Appellation EQ 'DP4753');
```

applied to the form shown in Figure 5.3(a) will produce the result shown in Figure 5.4(b).

Finally, two forms can be combined to create a third using the GRAFT operation. This operation is similar to the join operation defined by the relational model (Codd 1970). To illustrate, the following operation:

```
TEMP3 = GRAFT(Owners ONTO Parcel BEFORE Address)
          WHERE Owners.Appellation EQ Parcel.Appellation);
```

applied to the forms shown in Figure 5.3 produce the form shown in Figure 5.4(c).

Specification of a transfer using the languages DEFINE and CONVERT is converted by EXPRESS into a set of PL/1 Programs. These programs perform three translations:

Parcel					
	Address			Boundary	
Appellation	House	Street	City	Lines	Points
DP4753	13	Peverel	Christchurch	14	(5,10)
					(5, 20)
				15	(5, 10)
					(20, 10)
				19	(5, 20)
					(10, 25)
					(20, 20)
				16	(20, 20)
					(20, 10)
DP4972	15	Peverel	Christchurch	18	(20, 10)
					(40, 10)
				17	(40, 10)
					(40, 25)
				20	(20, 20)
					(40, 25)
				16	(20, 20)
					(20, 10)

(a) The Parcel form

Owners	
Appellation	Owner-name
DP4753	P. H. Davidson
DP3412	M. S. Cuthbert

(b) The Owner form

Figure 5.3: Examples of forms

1. *read*, performed by a program generated as follows:

$$\begin{array}{ccc} & \text{DEFINE} & \\ & \text{spec.} & \\ & \Downarrow & \\ \mathcal{S}(C_S,\ I_S,\ P_S) & \xrightarrow{read} & \mathcal{T}_0(C_S,\ I_S,\ P_0) \end{array}$$

The data set $\mathcal{S}$, conforming to the physical schema P_S, is transformed by a program generated from a DEFINE specification into the data set $\mathcal{T}_0$, conforming to the physical schema P_0.

2. *restructure*, which is performed by a program generated as follows:

TEMP1		
Appellation	Lines	Points
DP4753	14	(5, 10)
DP4753	14	(5, 20)
⋮	⋮	⋮
DP4753	16	(20, 10)
DP4972	18	(20, 10)
DP4972	18	(40, 10)
⋮	⋮	⋮
DP4972	16	(20, 20)
DP4972	16	(20, 10)

(a) The form TEMP1

TEMP2			
	Address		
Appellation	House	Street	City
DP4753	13	Peverel	Christchurch

(b) The form TEMP2

TEMP3						
	Owners	Address			Boundary	
Appellation	Owner-name	House	Street	City	Lines	Points
DP4753	P. H. Davidson	13	Peverel	Christchurch	14	(5,10)
						(5, 20)
					15	(5, 10)
						(20, 10)
					19	(5, 20)
						(10, 25)
						(20, 20)
					16	(20, 20)
						(20, 10)

(c) The form TEMP3

Figure 5.4: Forms created using operations provided by the CONVERT language

CONVERT

spec.

$$\Downarrow$$

$$\mathcal{T}_0(C_S,\ I_S,\ P_0) \xrightarrow{restructure} \mathcal{T}_1(C_D,\ I_D,\ P_1)$$

The data set $\mathcal{T}_0$, conforming to the source conceptual and implementation schemas C_S and I_S, is translated by a program generated from a CONVERT specification into the data set $\mathcal{T}_1$, conforming to the destination conceptual and implementation schemas C_D and I_D.

3. *write*, performed by a program generated as follows:

$$\begin{array}{ccc} & \text{DEFINE} & \\ & \text{spec.} & \\ & \Downarrow & \\ \mathcal{T}_1(C_D,\ I_D,\ P_1) & \xrightarrow{write} & \mathcal{D}(C_D,\ I_D,\ P_D) \end{array}$$

The data set $\mathcal{T}_1$, represented using forms according to the physical schema P_1, is translated by a program generated from a DEFINE specification into the final data set $\mathcal{D}$, conforming to the conceptual, implementation and physical schemas of the destination data representation.

The transfer $\mathcal{S} \mapsto \mathcal{D}$ performed by programs generated by EXPRESS is described as:

$$\begin{array}{ccccccc} & \text{DEFINE} & & \text{CONVERT} & & \text{DEFINE} & \\ & \text{spec.} & & \text{spec.} & & \text{spec.} & \\ & \Downarrow & & \Downarrow & & \Downarrow & \\ \mathcal{S}(C_S,\ I_S,\ P_S) & \xrightarrow{read} & \mathcal{T}_0(C_S,\ I_S,\ P_0) & \xrightarrow{restructure} & \mathcal{T}_1(C_D,\ I_D,\ P_1) & \xrightarrow{write} & \mathcal{D}(C_D,\ I_D,\ P_D) \end{array}$$

Taylor (1982) describes lessons learned during a five year involvement with the design, construction, and validation of EXPRESS. These lessons provide important insights into the general approach of using high-level data description and data manipulation techniques. Three examples of these insights are briefly described:

The importance of generality

A compromise has to be reached between providing a large and complex system that will process a wide variety of data, and a small and simple system that will process only a few types of data. The large and complex system may have a large 'learning curve', inhibiting the use of a system that may simplify the construction of translation software. The smaller and simpler system, on the other hand, may require additional software modules to be implemented by hand and, as a consequence, only marginally simplify the construction of translation software.

The importance of combined data description and data manipulation

Taylor remarks that 'a data description capability coupled with a data manipulation capability yields a much more powerful facility than simply a data description capability by itself'.

The importance of a methodology

The methodology developed for EXPRESS defines a procedure for designing and constructing data translation software that is based on representing data in forms, and on operations for manipulating these forms. Once familiar with this methodology, software to perform a translation could be constructed in a more compact and productive way.

5.1.2 SNAP

SNAP (Abbott 1989) is a more recent approach based on high-level data descriptions and data manipulations. Data processing is often accomplished using a combination of software tools that have been developed independently of each other and as a consequence use different data representations. Processing data using some combination of tools therefore requires the data to be transformed among the different representations used by the tools.

Applications to transform data are generated by SNAP from specifications of these transformations which are defined using Set Notation as A Programming Language (hence SNAP). Abbott (1989) illustrates the use of SNAP by explaining how an application was generated to transform data between the representation needed by P-Nut, a Petri-net analyser (Razouk 1987), and Teamwork, a data flow diagrammer (Cadre 1987).

Set notation 'is a declarative notation that combines some of the most useful features of functional programming and logic programming' (Abbott 1989). The structure of a SNAP program may be described using BNF as follows:

$$
\begin{array}{rcl}
\langle \text{SNAP program} \rangle & \longrightarrow & \langle \text{set definition} \rangle \mid \langle \text{structure definition} \rangle \\
\langle \text{set definition} \rangle & \longrightarrow & \langle \text{set name} \rangle = \{ \langle \text{set specification} \rangle \} \\
\langle \text{set specification} \rangle & \longrightarrow & \langle \text{element list} \rangle \\
 & \mid & \langle \text{generic element} \rangle \mid \langle \text{predicate} \rangle \\
 & \mid & \textit{file}\ \langle \text{file name} \rangle \\
\langle \text{element list} \rangle & \longrightarrow & \langle \text{element} \rangle \mid \langle \text{element list} \rangle\ , \langle \text{element} \rangle \\
\langle \text{structure definition} \rangle & \longrightarrow & \textit{structure}\ \{ \langle \text{structure component} \rangle \}
\end{array}
$$

Figure 5.5 contains an example SNAP program taken from Abbott (1989). Using this program, the set of values `one_ten = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }` are transformed into the set of values `{ 2, 4, 9, 16, 25, 36, 49, 64, 81, 100 }` by evaluating the expression `squared_elements(one_ten)`. Abbott suggests that SNAP would 'be quite appropriate' for transforming data to and from a representation such as that defined by the Initial Graphics Exchange Specification (IGES) (Smith & Wellington 1992).

```
squares(Source) = { X -> X*X | X in Source }
structure element -> square
squared_elements(Set) = { S.square | S in squares(Set) }
```

Figure 5.5: An example of a SNAP program(Abbott 1989)

In general, a transformation of data set $\mathcal{S}$ into the destination data set $\mathcal{D}$ by an application called *appl* generated by SNAP can be described as:

$$\begin{array}{ccc} & \text{SNAP} & \\ & \text{specification} & \\ & \Downarrow & \\ \mathcal{S} & \xrightarrow{appl} & \mathcal{D} \end{array}$$

Although useful, SNAP has limited facilities for processing data represented in files. Transformation applications generated by SNAP can read files containing data values that are represented only as Prolog terms. Although representation of the transformed data values produced by the application is unspecified, the author expects these values to be also represented as Prolog terms in a file.

The SNAP notation for specifying data transformations 'is an extension of (domain) relational calculus' which 'has as much expressive power as any relational database language' and 'provides flexibility beyond that available in most database systems, i.e. it is not limited to the *flat file* databases' (Abbott 1989). This limitation of relational database systems which Abbott refers to has, however, recently been reduced by the development of systems like Postgres (StoneBraker 1992) which provides a query language for processing relations that may have structured multi-valued attributes.

5.1.3 The Chameleon Project

In the Chameleon Project, Mamrak *et al* (1989) addressed the problem of constructing interfaces for transforming electronic manuscripts among various non-standard representations. These non-standard representations are those required by word processing packages such as Scribe (Unilogic 1984), troff (Barron & Rees 1987), or LaTeX (Lamport 1986).

The transfer of an electronic manuscript $\mathcal{S} \mapsto \mathcal{D}$ is accomplished by mapping the source electronic manuscript $\mathcal{S}$ in some non-standard representation, into some standard form $\mathcal{E}$, and then onto the destination non-standard representation $\mathcal{D}$ of this manuscript. The general form of a manuscript transfer is described as:

$$\mathcal{S} \xrightarrow{up} \mathcal{E} \xrightarrow{down} \mathcal{D},$$

where $\mathcal{S} \xrightarrow{up} \mathcal{E}$ is an *up-translation* performed on the non-standard representation $\mathcal{S}$ by an *up-translator*; and $\mathcal{E} \xrightarrow{down} \mathcal{D}$ is a *down-translation* performed on the non-standard representation $\mathcal{D}$ by a *down-translator*.

Figure 5.6 is an example of a manuscript transfer taken from Mamrak *et al* (1989).

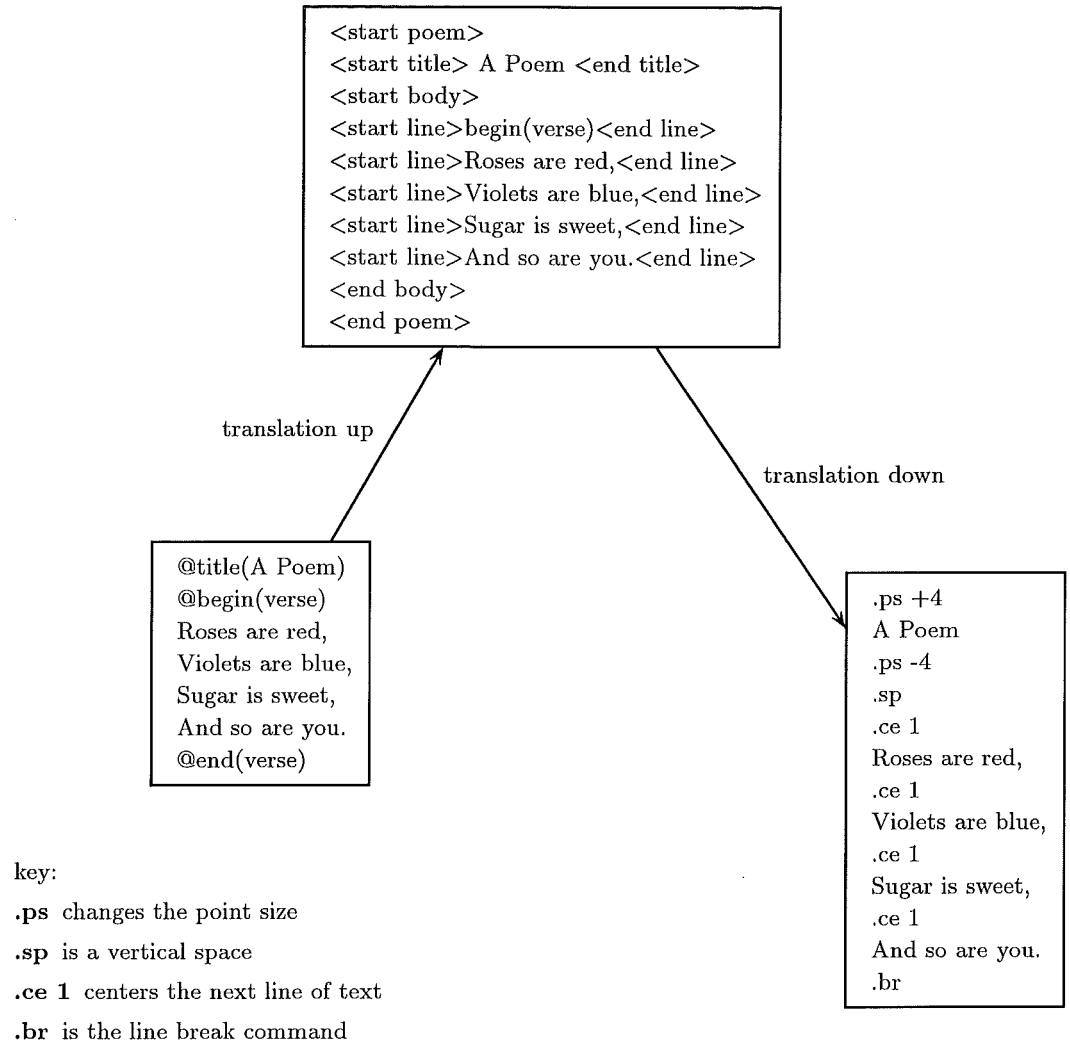


Figure 5.6: An example taken from Mamrak *et al* (1989) of a manuscript translation using the Chameleon approach

The source representation is that required by Scribe (Unilogic 1984), and the destination representation is that required by troff (Barron & Rees 1987).

The general form of a manuscript transfer $\mathcal{S} \rightarrow \mathcal{E} \rightarrow \mathcal{D}$ is similar to that of a geographical data transfer $\mathcal{S} \rightarrow \mathcal{I} \rightarrow \mathcal{D}$ governed by the interchange format interfacing strategy, as discussed in Section 3.4. The standard form of a manuscript $\mathcal{E}$ corresponds to the geographical data set $\mathcal{I}$, represented according to a standard interchange format.

Various standard forms may be defined for electronic manuscripts. These definitions are called Document Type Definitions (DTD) and examples are *thesis* and *article*. As discussed in Section 3.4.2, the complexity of the interchange format defined by the Spatial Data Transfer

Standard (SDTS) (Geological Survey 1992), for example, has resulted in the definition of smaller formats, called *profiles*, that are self-contained subsets of the SDTS. These profiles are analogous to Document Type Definitions.

Any DTD is likely to be less complex and more precise than any comprehensive interchange format for geographical data because any DTD is for a single type of electronic manuscript. In contrast, an interchange format for geographical data has to represent a wide variety of data types. Constructing up- and down- translators for transforming manuscripts to and from DTDs is therefore likely to be easier than constructing interfaces for translating geographical data to and from a representation defined by an interchange format.

Construction of a down-translator can be described as:

$$\begin{array}{ccc} & DTD & \\ & \Downarrow & \\ & \underbrace{TD_S \;\; SFPD \;\; SFSD} & \\ & \Downarrow & \\ \mathcal{E} & \xrightarrow{down-translator} & \mathcal{S} \end{array}$$

where

DTD

is the document type definition which is processed to produce another specification, the Standard Form Parser Definition (SFPD). The SFPD is processed by `yacc` (Johnson 1979) to generate a parser for reading the manuscript;

SFSD

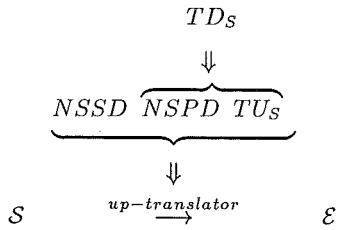
is the Standard Form Scanner Definition which specifies the tokens within the standard form representation of the manuscript. Examples of these tokens are: opening and closing brackets, and data values. The SFSD is processed by `lex` (Lesk & Schmidt 1979) to generate a scanner for recognising these tokens and passing them onto the parser;

TD_S

which specifies the mapping of a manuscript from a standard representation $\mathcal{E}$ into the non-standard representation $\mathcal{S}$.

Techniques used by Mamrak *et al* (1989) allow the mapping TD_S to be used in the

construction of an up-translator, described as follows:



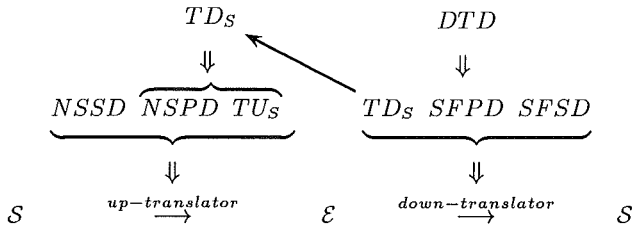
where

NSSD
: is the Non-Standard Scanner Definition which specifies the tokens within the non-standard form representation of the manuscript;

NSPD
: is the non-standard parser definition generated from the down-translation specification TD_S. The NSPD is processed by `yacc` to generate a parser for reading the non-standard representation of the manuscript;

TU_S
: defines the mapping of the manuscript from the non-standard representation $\mathcal{S}$ into the standard representation $\mathcal{E}$.

Constructing the up- and down- translators for some non-standard representation of a manuscript $\mathcal{D}$ can be described as:



The Chameleon approach to interface construction is similar to the author's, described in Chapter 6, for constructing geographical interfaces. These similarities are: the use of software tools `yacc` and `lex` to generate software for reading the text file representation of the data values; and the generation of interfaces from high-level definitions of the data transfers these interfaces are to perform.

The Chameleon approach differs from the author's in that data is transformed from the source representation to the destination representation through some intermediate representation, called the standard form. The author's approach is more similar to that of

the previously described approaches of EXPRESS and SNAP where the data is transformed directly from the source representation to the destination representation without some transformations to and from an intermediate representation.

Although the Chameleon Project is completed, research into translation of many kinds of data is being continued in the Nagiya Project.

5.1.4 The Nagiya Project

The objective of the Nagiya project is to 'support the automatic generation of translators between different types of applications and fundamentally different data models' (Gawkowski & Mamrak 1992). Within this research project, a universal framework for data translation has been proposed.

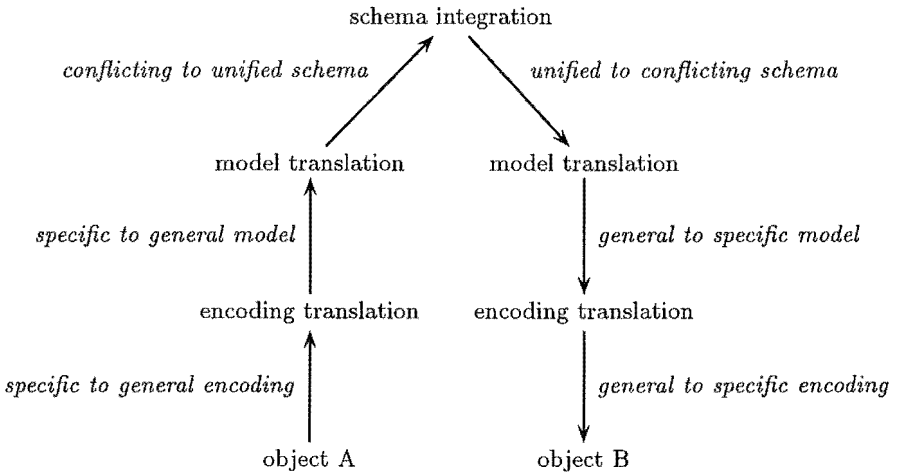


Figure 5.7: The proposed universal architecture for data translation(Gawkowski & Mamrak 1992)

The proposed framework, shown in Figure 5.7, divides a data translation into three general processes, which are described below.

Process 1: Transforming data between a general and a specific encoding

Transforming data values between a specific encoding in a specific data model, and a general encoding in a specific data model. An example given by Gawkowski & Mamrak (1992) is that of translating data sets with ASCII representations peculiar to relational database products like Oracle, BASICplus, and DBase, into data sets having the same representation for fields, records, and tables.

Using the notation defined in Chapter 3, the transformation from a specific encoding to a general encoding can be described as:

$$\mathcal{S}(C_S,\ I_S,\ P_S) \rightarrow \mathcal{T}_0(C_S,\ I_S,\ P_0)$$

where

P_S is the physical schema which defines an encoding or physical representation of the source data values $\mathcal{S}$ using the constructs specified by an encoding model specific to the source geographical information system; and

P_0 is the physical schema which defines an encoding or physical representation of the source data values $\mathcal{T}_0$ using the constructs specified by a general model for representing data values in perhaps a text file.

An example of this transformation is the translation of data values, represented in a text file according to the Gina file format (GeoVision 1986), into a data set also represented in a text file, but according to the general model defined by ISO 8211.

Process 2: Transforming data between a specific data model and a general model

Transforming data values between a specific data model with a specific schema, and a general model with a specific schema. An example given by Gawkowski & Mamrak is:

> 'some manuscripts may be modelled as a set of hierarchical relationships among objects, i.e., a grammar. Alternatively, some databases are modelled as tables, records, and fields. This translation step would remodel manuscripts and databases using a general model in which sections, chapters, tables, and so on, are represented using a common scheme, independent of the semantics of grammar or table-based models' (Gawkowski & Mamrak 1992)

The transformation of data values from a specific data model to a general model can be described as:

$$\mathcal{T}_0(C_S,\ I_S,\ P_0) \rightarrow \mathcal{T}_1(C_S,\ I_G,\ P_G)$$

where

I_S is the implementation schema that defines the types of data in $\mathcal{T}_0$ using the types of data specfied by the implementation model specific to some source geographical information system;

I_G is the implementation schema that defines the types of data in $\mathcal{T}_1$ using the types of data specfied by a general implementation model. Examples of such models may be the spatial data model defined by the Spatial Data Transfer Standard (Geological Survey 1992), or the generic model for planar geographical objects proposed by Worboys (1992); and

P_G is a general encoding of the data values appropriate for the types defined by the general implementation model.

Process 3: Transforming data between conflicting and unified schemas

Transforming data values between a general model with a specific schema, and a general model with a general schema. This third process 'involves translating from conflicting schemas to a unified schema within the general data model'. In the context of an object-oriented model, examples of the conflicts that may have to be resolved include:

- an object type in one schema may be represented as a relationship in another schema,
- a collection of object types in one schema may be represented by a single object in another schema, or
- the object types may have different names in different schemas.

This transformation can be described as:

$$\mathcal{T}_1(C_S,\ I_G,\ P_G) \rightarrow \mathcal{U}(C_G,\ I_G,\ P_G)$$

where

C_S is the conceptual schema that defines those aspects of the real world phenomena for which data values were captured according to the requirements of the user of the source data.

C_G is the conceptual schema that defines the same aspects as those defined by C_S, but expressed using standard terminology.

An example of renaming spatial phenomena to conform to standard terminology might be the mapping of local terms such as a street or an accessway onto a term road, defined by the Spatial Data Transfer Standard as being 'an open way for the passage of vehicles, persons, or animals on land' (Geological Survey 1992).

Software tools are being developed as part of the Nagiya Project for supporting this universal framework for data translation.

5.1.5 Summary

All the different approaches to contructing general interfaces described in Section 5.1, excluding the Nagiya Project because of a lack of information, are summarised in Table 5.1. This Table is divided into two parts: one summarising the approaches to general interface construction, and the other summarising the research performed by the author into geographical interface construction. The latter includes earlier research by the author, to be described in Section 5.2.4, and the author's approach described in this thesis.

The approaches are characterized by the general form of the transfer performed by the interfaces constructed using these approaches. Use of EXPRESS and SNAP produces interfaces that perform transfers having the general form associated with the individual interfacing strategy. Use of the Chameleon and Nagiya projects produce interfaces that perform transfers having the general form associated with the interchange interfacing strategy.

The EXPRESS and SNAP approaches also differ from those of the Chameleon and Nagiya Projects in the amount of assistance given during the construction of interfaces. The EXPRESS and SNAP approaches provide languages in which all the necessary data translations have to be specified by the constructor of the interface. These high-level specifications are turned into programs that perform the specified translations.

The Chameleon approach requires the specification of the down-translation to be given in a form that allows a software tool to invert this specification to produce a specification of the up-translation. Software can then be generated for performing both the up- and down-translations between a standard form and some other representation of an electronic manuscript.

Having discussed various approaches to interface construction for transferring data in general, the remainder of this Chapter contains a description of research into different approaches for constructing interfaces to transfer geographical data.

5.2 Geographical interfaces

Traditionally, geographical interfaces are constructed by programmers who read the definitions of the source and destination geographic data representations, and then implement these interfaces. Usually these interfaces are implemented in some widely used language such as C (Kernighan & Ritchie 1978). A change to either the source or destination representation requires the interface to be modified and, after several changes, the interface may have to be implemented again. This Section is a review of various approaches that were developed to simplify the time-consuming and complex task of interface construction.

Approach	Source Data Representation		Data Translation Environment		Destination Data Representation	
	Definition Language	Representation	Definition Language	Representation	Definition Language	Representation
EXPRESS	DEFINE	Hierarchical data file	CONVERT	Hierarchical data structures that are viewed as forms	DEFINE	Hierarchical data file
SNAP	None	A file containing Prolog terms	SNAP	Set theory incorporating a combination of functional and logical programming concepts	None	A file containing Prolog terms
Chameleon	Grammars	A file containing data represented as a string of the language described by a grammar	Unknown	Attribute grammars and parse trees	Grammars	A file containing data values represented as a string of the language described by a grammar

For comparison, two approaches developed by the author to geographical interface construction are presented below

'divide and conquer'	Grammars	A file containing data represented as a string of the language described by a grammar	Quel	Relational data model	None	None
Current approach	a2b	A file containing data represented as a string of the language described by a grammar	C or Miranda	C or Miranda data structures	a2b	A file containing data represented as a string of the language described by a grammar

Table 5.1: A summary of the approaches to constructing interfaces

5.2.1 GEOLINK

The GEOLINK system (Waugh & Healy 1986) is a general purpose interfacing tool for transforming data produced by one system into a representation that can be processed by another. Data may be transformed either by:

- a user providing GEOLINK with two input files: a *spool file* comprising the data values to be transformed; and a *mask file* comprising GEOLINK instructions for processing the spool file, and any other data to be included in the file generated by GEOLINK. GEOLINK merges the spool and mask files to generate the *target file*;

- a user providing GEOLINK with data in response to questions defined within the mask file. The functionality of the GEOLINK commands is sufficient to allow 'complete menu or conversational systems $\cdots$ [to] be set up which need not, in fact, use a spool file' (Waugh & Healy 1986); or

- some combination of the above.

An example of a spool file, a mask file, and the generated target file, taken from Waugh & Healy (1986), is presented in Figure 5.8. The spool file comprises three sets of data values for the months of January through to May, and the mask file comprises a mixture of GEOLINK commands (prefixed by the character '!') and GIMMS instructions. GIMMS is a geocartographic processing system developed by Waugh & McCalden (1983). The target file generated by GEOLINK will produce three graphs when executed by GIMMS. The general form of this transfer can be described as:

$$\mathcal{S}(C_S, I_S, P_S) \xrightarrow[\Downarrow \text{ the mask file}]{GEOLINK} \mathcal{D}(C_D, I_D, P_D)$$

where $\mathcal{S}(C_S, I_S, P_S)$ corresponds to the spool file, $\mathcal{D}(C_D, I_D, P_D)$ corresponds to the target file, and the definition contained within the mask file is used by GEOLINK to produce the target file.

5.2.2 Spatial Data Research and Support System

The Spatial Data Research and Support System (SDRSS) (van Roessel *et al* 1986) comprises a collection of geographical information systems for storing and processing diverse spatial data, and a collection of interfaces for translating data among the different representations used by the various geographical information systems within the SDRSS. SDRSS provides

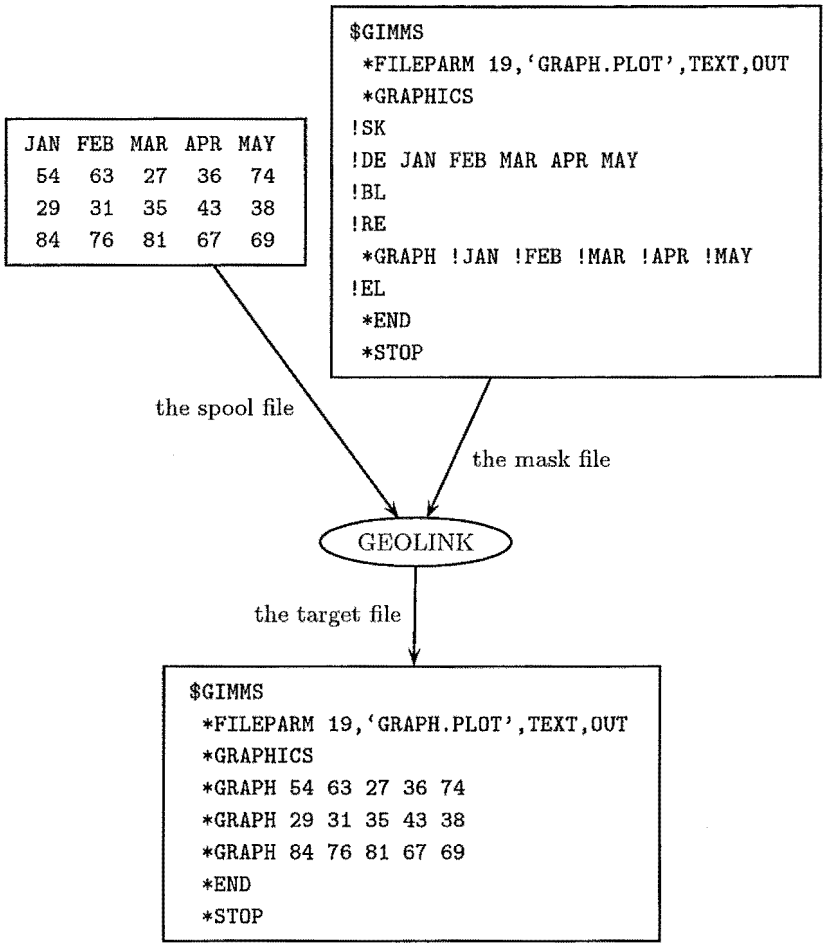


Figure 5.8: An example of using GEOLINK

'an integrated set of system resources to support data acquisition, storage, processing, analysis, and product generation requirements of a broad research program directed toward the integration and application of disparate spatial data types' (van Roessel *et al* 1986).

Each geographical information system included within SDRSS has a collection of interfaces that are divided into two groups: *input interfaces*, which translate data from various representations into the representation used by the geographical information system; and *output interfaces*, which translate data from the representation used by the geographical information system into various representations used by other geographical information systems.

As an example, van Roessel *et al* (1986) describes pairs of input and output interfaces for translating data between the representation required by ARC/INFO, and representations

required by other systems including: RIM, the Relational Information Management system, in which data is represented using relations comprising single and multi-valued attributes; and UCGLES, the Unified Cartographic Line Graph Encoding System developed by the USGS National Mapping Division, in which data is represented as Digital Line Graphs (DLGs) as defined by Geological Survey (1990).

van Roessel *et al* (1986) developed a methodology for constructing interfaces that are structured in a manner similar to a model proposed by the Workgroup on Data Organization of the National Committee for Digital Cartographic Data standards (Nyerges 1984). The proposed model for data translation has a general form similar to that defined by the interchange interfacing strategy that was discussed in Section 3.4.2. Using the notation developed in Section 3.2, the proposed model is described as:

$$\mathcal{S} \xrightarrow{t_1} \mathcal{I}_1 \xrightarrow{t_2} \mathcal{I}_2$$

where

$\mathcal{S}$ is the data set to be transferred;

$\xrightarrow{t_1}$ is a transformation of the source data set into some intermediate data set $\mathcal{I}_1$;

$\mathcal{I}_1$ is some intermediate data set;

$\xrightarrow{t_2}$ is a transformation of the data set into an interchange data set $\mathcal{I}_2$; and

$\mathcal{I}_2$ is a data set analogous to the data set $\mathcal{I}$ which is created during a transfer $\mathcal{S} \mapsto \mathcal{I} \mapsto \mathcal{D}$ according to the interchange format interfacing strategy.

The above model is for only one half $\mathcal{S} \xrightarrow{*} \mathcal{I}_2$ of the complete transfer $\mathcal{S} \xrightarrow{*} \mathcal{I}_2 \xrightarrow{*} \mathcal{D}$; the second half, according to van Roessel *et al* (1986), is similar to the first.

An interface first applies the transformation t_1 to the original data set $\mathcal{S}$ to produce the set $\mathcal{I}_1$ comprising of relations having multi-valued attributes within RIM. The transformation t_2 transforms the data set $\mathcal{I}_1$ into the set $\mathcal{I}_2$ comprising data represented using the set of relations shown in Figure 5.9. Use of the relational model for translating geographical data is discussed further in Section 5.2.3.

Use of BNF for 'concisely and consistently' (van Roessel *et al* 1986) describing the various data representations used by different geographical information systems was another aspect to the SDRSS methodology for constructing interfaces. As will be described in Section 5.2.4, BNF descriptions can be adapted for use by software tools such as `yacc` to generate the part of an interface that reads data represented in a text file.

Relation name	Attributes
regpol:	regnum, polnum
polarc:	polnum, arcnum
archdr:	arcnum, strtnode, endnode, lftreg, rgtreg
arcxy:	arcnum, x, y
nodearc:	nodenum, arcnum
nodexy:	nodenum, x, y

Figure 5.9: The relations comprising the I2 structure

5.2.3 Relational methods for translating geographical data

van Roessel & Fosnight (1985), van Roessel *et al* (1986), and Penny (1986) suggest using the structures and operators provided by the relational data model for transforming geographical data between different representations. The SDRSS methodology for interface construction described in the preceding Section is a practical example of using relational concepts for data translation. Another example is to be discussed in Section 5.2.4.

van Roessel & Fosnight (1985) advocated the use of the relational model for three reasons:

1. elegance and simplicity of the data representation,
2. the availability of the relational algebra and its unique relational operators, and
3. the availability of a number of different relational database management systems on various hardware configurations.

Penny (1986) concluded that 'particularly effective tools $\cdots$ should be obtainable by starting with the fundamental relational idea, $\cdots$, but setting out to develop a set of operations designed expressly for manipulating spatial data'. This conclusion is substantiated by the SDRSS approach to data translation (van Roessel *et al* 1986), in which the transformation $\mathcal{I}_1 \xrightarrow{t_2} \mathcal{I}_2$, was accomplished by a mixture of: relational operators such as `JOIN`, `PROJECT`, and `INTERSECT` provided by RIM; and independent programs for performing the following classes of tasks: the input of data into RIM; general and topological data restructuring; coordinate conversion; and display. In earlier research, the author used the Ingres (Date 1990) relational DBMS for translating geographical data.

5.2.4 Earlier research

Pascoe (1989) proposed a 'divide and conquer' approach to constructing interfaces for transforming the text file representation of values that were moved between computer systems

using magnetic tape. Use of communication networks by interfaces for moving data values was not considered. A model interface was defined that divided an interface into smaller modules which were either constructed using existing software tools, or used high-level operators and data structures provided by the relational data model (Codd 1970). This approach and a practical example are described next.

5.2.4.1 Model interface

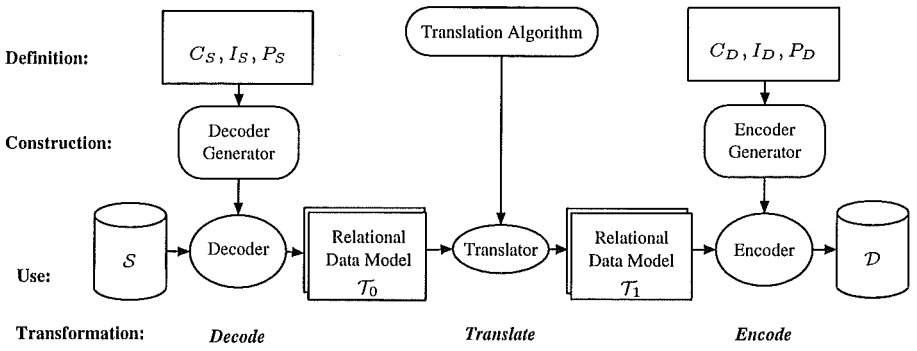


Figure 5.10: A Model Interface

As shown in Figure 5.10, a model interface was defined to consist of three modules: a *decoder,* which reads data from the source file representation; a *translator,* which transforms the source data values into the types of values required for the destination geographical information system; and an *encoder,* which writes the translated data values into the destination file representation.

Using the notation presented in Section 3.2, a transfer $\mathcal{S}(C_S,\ I_S,\ P_S) \mapsto \mathcal{D}(C_D,\ I_D,\ P_D)$ comprises 3 transformations:

decode

$$\mathcal{S}(C_S,\ I_S,\ P_S) \stackrel{decode}{\longrightarrow} \mathcal{T}_0(C_S,\ I_S,\ P_0)$$

Data values physically represented according to the schema P_S are decoded and placed into a temporary representation defined by P_0;

translate

$$\mathcal{T}_0(C_S,\ I_S,\ P_0) \stackrel{translate}{\longrightarrow} \mathcal{T}_1(C_D,\ I_D,\ P_1)$$

The source data values are translated to conform to the conceptual and implementation schemas defined for the destination geographical information system. The physical representation is transformed accordingly; and

encode

$$\mathcal{T}_1(C_D,\ I_D,\ P_1) \stackrel{encode}{\longrightarrow} \mathcal{D}(C_D,\ I_D,\ P_D)$$

The translated data values physically represented according to the schema P_1 are encoded into the destination physical representation defined by P_D.

The transfer $\mathcal{S} \mapsto \mathcal{D}$ performed by the model interface can therefore be described as:

$$\mathcal{S}(C_S,\ I_S,\ P_S) \stackrel{decode}{\longrightarrow} \mathcal{T}_0(C_S,\ I_S,\ P_0) \stackrel{translate}{\longrightarrow} \mathcal{T}_1(C_D,\ I_D,\ P_1) \stackrel{encode}{\longrightarrow} \mathcal{D}(C_D,\ I_D,\ P_D).$$

To illustrate this transfer process, consider the transfer of a data set $\mathcal{S}$, which conforms to a simplified form of the Gina file format (GeoVision 1986), into the data set $\mathcal{D}$. The data set $\mathcal{D}$ conforms to a simplified form of the text file representation required for processing by the Grass geographical information system (Shapiro *et al* (1992)). The data set $\mathcal{S}$ shown in Figure 5.11(a) is decoded into the data set $\mathcal{T}_0$ shown in Figure 5.11(b). The data set $\mathcal{T}_0$, represented within a relational database, is translated into the data set $\mathcal{T}_1$ shown in Figure 5.11(c). Finally, the data set $\mathcal{T}_1$ is encoded into the data set $\mathcal{D}$ shown in Figure 5.11(d).

Construction of the interface that performs this transfer is discussed in the next Section, and a listing of the source code for this interface is presented in Appendix B. For comparison, construction of an equivalent interface using the author's software tool `a2b`, to be described in later Chapters, is also presented in Appendix B.

Dividing an interface into the three transformations: decode, translate, and encode, reduces the effort of constructing two or more interfaces involving any one geographical data set $\mathcal{B}(C_B, I_B, P_B)$. Given the implementation of interfaces to perform the transfers $\mathcal{A} \mapsto \mathcal{B}$ and $\mathcal{B} \mapsto \mathcal{C}$, a new interface $\mathcal{A} \mapsto \mathcal{C}$ will require the construction of only one module to perform the translate transformation because the modules performing the respective decode and encode transformations from the interfaces $\mathcal{A} \mapsto \mathcal{B}$ and $\mathcal{B} \mapsto \mathcal{C}$ can be reused in the new interface $\mathcal{A} \mapsto \mathcal{C}$.

This is illustrated in Figure 5.12, where the three data sets $\mathcal{A}, \mathcal{B}$, and $\mathcal{C}$ are transferred using interfaces that are collectively implemented by 3 encoders, 3 decoders, and 6 translators. A complete set of interfaces for an interchange group comprising N geographical information systems which transfer data according to the individual interfacing strategy would require N decoders, N encoders, and $N^2 - N$ translators.

```
feat 874 L
coor 867 543 874 550 880 564
feat 875 L
coor 880 564 876 580
feat 876 P
coor 872 550
```

(a) The data set $\mathcal{S}$

feature	
id	*ftype*
874	L
875	L
876	P

coor			
id	*seq*	*x*	*y*
874	0	867	543
874	1	874	550
874	2	880	564
875	0	880	564
875	1	876	580
876	0	872	550

(b) The data set $\mathcal{T}_0$

line definition			
id	*seq*	*x*	*y*
874	0	867	543
874	1	874	550
874	2	880	564
875	0	880	564
875	1	876	580

line description					
id	*npts*	*mnx*	*mny*	*mxx*	*mxy*
874	3	867	543	880	564
875	2	876	564	880	580

point		
id	*x*	*y*
876	872	550

(c) The data set $\mathcal{T}_1$

```
Line 874 3 867 543 880 564
867 543
874 550
880 564
Line 875 2 876 564 880 580
880 564
876 580
Point 876 872 550
```

(d) The data set $\mathcal{D}$

Figure 5.11: An example transfer showing the data values at different stages of different transformation

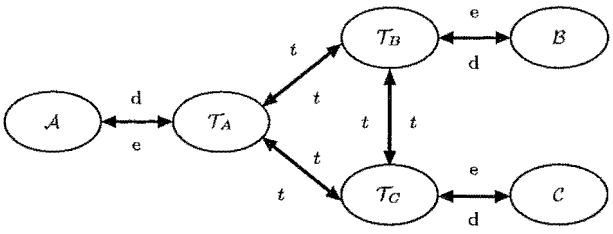


Figure 5.12: Reusing the modules of an interface

5.2.4.2 Generating interfaces

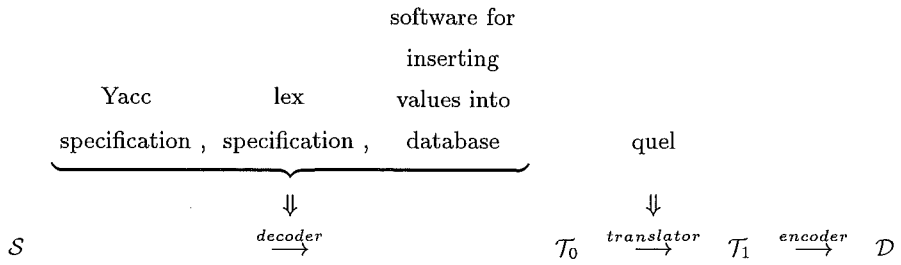
The author's objective for the methodology just described was to simplify interface construction either by using existing software tools to generate components of the interface modules for transforming the data, or by using operators provided by other software packages for transforming the data. Construction of decoder, translate, and encoder modules of an interface module are briefly described below.

A decoder module is implemented by software generated by the tools `yacc` and `lex` for reading the data values represented in a text file, and software provided by the person constructing the decoder for placing these data values into a set of relations within a relational database managed by Ingres.

A translator is implemented by a sequence of relational operations expressed using the `quel` query language provided by the relational database system Ingres (Date 1990). Relational operators are applied to the source data relations to produce data within other relations having the general form required for the destination geographical information system. Any transformations better performed by operations not provided by the query language could be implemented as C functions that accessed the data stored within the relational database. A translator could therefore be a combination of C functions and functions comprising embedded `quel` operations.

An encoder is constructed manually because no existing software tool was found to provide any assistance for constructing software that retrieved data values from a relational database, and encoded them into a text file representation.

Constructing an interface can in general be described as:



where the decoder module is constructed of software generated from `yacc` and `lex` specifications, and of routines constructed manually for inserting the data into an Ingres database; and the translator and encoder are constructed manually.

To illustrate this process, consider the construction of an interface to perform the transfer shown in Figure 5.11. The decoder was generated from `yacc` and `lex` input files, parts of which are shown in Figure 5.13. A translator module was made up of the sequence of relational operations shown in Figure 5.14. The encoder module is presented in Appendix B.1, together with a complete listing of this interface.

The next Chapter contains a description of extensions made by the author to this earlier research, including: generating an encoder and a decoder from a single specification of a text file data representation; and constructing interfaces that send data through a communications network using ISO-OSI communication protocols.

```
%union{ char *string;   int integer; }
%token FEAT COOR
%token <string> STRING
%token <integer> INTEGER
%%
file        : feature | file feature ;
feature     : FEAT description COOR coordinates ;
description : INTEGER STRING { appendFeature( $1, $2); };
coordinates : INTEGER INTEGER { appendCoor($1, $2); }
            | coordinates INTEGER INTEGER { appendCoor($2, $3); };
%%
#include "decoderLex.c"
decode(filename) char *filename;
{
    freopen(filename, "r", stdin);
    yyparse();
}
```

The yacc input file

```
D           [0-9]
CHAR        [A-Za-z]
%%
feat        { return FEAT; }
coor        { return COOR; }
{CHAR}+     { dupstr(yytext,&((*yylvalp).string)); return STRING; }
{D}+        { (*yylvalp).integer = atoi(yytext); return INTEGER; }
.|\n            /* ignore */
```

The lex input file

Figure 5.13: Parts of the Yacc and lex input files from which was generated a decoder for the example interface

```
retrieve linedefinition(id = coor.id, seq = coor.seq,
                        x = coor.x, y = coor.y)
  where coor.id = feature.id and feature.ftype = "L"
retrieve point(id = coor.id,x = coor.x, y = coor.y)
  where coor.id = feature.id and feature.ftype = "P"
retrieve linedescription(
               id = coor.id, npts = count(coor.seq by coor.id),
               mnx = min(coor.x by coor.id),
               mny = min(coor.y by coor.id),
               mxx = max(coor.x by coor.id),
               mxy = max(coor.y by coor.id))
   where count(coor.seq by coor.id) > 1
```

Figure 5.14: The relational operations forming the translator for the example interface

Approach to constructing geographical interfaces

Chapter 6

Chapters 6 to 11 contain a description of the author's approach to constructing geographical interfaces. These interfaces either: move data from one computer system to another through a communications network; transform the text file representation of data; or both move and transform data.

The overall approach to interface construction is to generate interfaces from formal definitions, called transfer specifications, of the data transfers these interfaces are to perform. Transfer specifications and a notation for defining them are discussed in Chapter 7. The author has developed the software tool `a2b` for simplifying interface construction by generating interfaces from a transfer specification. Interface construction using `a2b` is discussed in Chapters 8 to 11. The topic for each Chapter is:

Chapter 8
: generating the main routine of an interface from a transfer specification;

Chapter 9
: generating the encoder and decoder modules from a transfer specification;

Chapter 10
: constructing translator modules from a transfer specification; and

Chapter 11
: generating communicator modules from a transfer specification.

An interface is divided into smaller self-contained *interface modules*, which are executed in sequence according to a *main interface routine*. In Section 6.1 four types of interface module

are defined: encoders, decoders, translators, and communicators. Each type of interface module performs a different data transformation and rules for combining these different interface modules to form interfaces are described in Section 6.2.

In proposing this approach, the author has sought to simplify the construction of interfaces without compromising the quality of the transfer performed by these interfaces. That is, to construct interfaces which do not lose information for the reasons discussed in Section 3.2.1. As was shown in Section 3.4.1, the individual interfacing strategy would be the best interfacing strategy, were it not for the large number of interfaces that have to be constructed. One goal of the author has therefore been to simplify interface construction to the point where the difficulty of constructing interfaces need not be a concern when comparing different interfacing strategies.

6.1 Interface modules

An interface is divided into four types of interface module: decoders, encoders, translators, and communicators. Each module performs one type of data transformation as will be described in the next four Sections.

6.1.1 Decoder modules

A *decoder module* transforms the physical representation of the data values. Using the notation defined by the author in Chapter 3, this transformation can be described as:

$$\mathcal{S}(C_S,\ I_S,\ P_S) \xrightarrow{decode} \mathcal{T}_0(C_S,\ I_S,\ P_0).$$

As was discussed in Section 4.4, transformations in general may be viewed as either application layer transformations, presentation layer transformations, or transformations at the other 5 lower layers of the ISO-OSI reference model. Two types of decoder are therefore defined:

an application layer decoder

transforming data having a text file representation into data having a memory-resident representation using C data structures; and

a presentation layer decoder

transforming data having a memory-resident representation, suitable at the presentation layer, into data having a memory-resident representation suitable for use at the application layer.

6.1.2 Encoder modules

An *encoder module* also transforms the physical representation of the data values, and is described as:

$$\mathcal{T}_N(C_D,\ I_D,\ P_N) \stackrel{encode}{\longrightarrow} \mathcal{D}(C_D,\ I_D,\ P_D).$$

As was the case with decoders, two types of encoder are defined:

an application layer encoder
: transforming data having a memory-resident representation using C data structures into data having a text file representation; and

a presentation layer encoder
: transforming data having a memory-resident representation suitable for use at the application layer into data having a memory-resident representation suitable for use at the presentation layer.

An application layer encoder and an application layer decoder perform a complementary pair of transformations such that:

$$\mathcal{S} \stackrel{decode}{\longrightarrow} \mathcal{T} \stackrel{encode}{\longrightarrow} \mathcal{S}.$$

Similarly, a presentation layer encoder and a presentation layer decoder perform a complementary pair of transformations such that:

$$\mathcal{T}_S \stackrel{encode}{\longrightarrow} \mathcal{N}_S \stackrel{communicate}{\rightsquigarrow} \mathcal{N}_D \stackrel{decode}{\longrightarrow} \mathcal{T}_D$$

where:

- the data sets $\mathcal{T}_S$, $\mathcal{T}_D$, $\mathcal{N}_S$, and $\mathcal{N}_D$ have the same conceptual and implementation schemas;
- the sets $\mathcal{N}_S$, and $\mathcal{N}_D$ are identical but are located on different computer systems; and
- the sets $\mathcal{T}_S$ and $\mathcal{T}_D$ have physical representations that differ due to any disparity in hardware between the two computer systems.

6.1.3 Translator modules

A *translator module* translates data values to conform to a different conceptual schema, a different implementation schema, or both. A translator that transforms a set of data values

$\mathcal{T}_0$ into another data set $\mathcal{T}_1$, conforming to different conceptual and implementation schemas, is described as:

$$\mathcal{T}_0(C_0,\ I_0,\ P_0) \overset{*}{\rightarrow} \mathcal{T}_1(C_1,\ I_1,\ P_1).$$

Note that transforming a data set to conform to different conceptual or implementation schemas also results in the transformed values having a different physical representation.

Translator modules are regarded by the author as performing only application layer transformations, that is, data transformations between different views of data at either or both the conceptual and implementation levels of abstraction. Examples of these transformations are: mapping street addresses onto legal property identifiers; and transforming values between different geometric models.

6.1.4 Communicator modules

A *communicator module* moves the data values from a source computer system to a destination computer system through a communications network. This transformation is described as:

$$\mathcal{N}_S(C_N,\ I_N,\ P_N,\ L_S) \overset{communicate}{\rightsquigarrow} \mathcal{N}_D(C_N,\ I_N,\ P_N,\ L_D)$$

Any communicator performs the data transformations occurring at the other 5 layers of the ISO-OSI reference model.

6.2 Combining modules to form interfaces

An interface comprises a combination of decoder, encoder, translator, and communicator modules according to the required data transfer. Rules which govern the manner in which the four types of modules may be combined are described below.

6.2.1 Combining communicator modules

A communicator module moves a set of data values conforming to a network representation through a communications network. A communicator module must be preceded by a presentation layer encoder which transforms data into a set of values conforming to the physical schema of a network representation. This must be followed by a presentation layer decoder which transforms data conforming to the physical schema of a network representation into a set of values conforming to another physical schema. This combination of modules is described as:

$$\begin{array}{lll} \text{Presentation} & \mathcal{T}_i \overset{encode}{\longrightarrow} \mathcal{N}_S & \mathcal{N}_D \overset{decode}{\longrightarrow} \mathcal{T}_{i+1} \\ \text{Layers 1–5} & \mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D & \end{array}$$

As will be explained in Chapter 11, communicator modules and their associated presentation layer decoders and encoders are constructed by the author's software tool `a2b` using software tools and run-time libraries provided by ISODE, described earlier in Section 4.5.

6.2.2 Combining translator modules

As will be explained in Chapter 7, a translator can be defined within a translation environment provided by either the C programming language (Kernighan & Ritchie 1978), or the Miranda functional programming language (Turner 1986). Before data values can be translated in either environment, values must be transformed using a combination of encoder and decoder modules to have a physical representation suitable for the environment within which the translator is defined.

A translator defined within the C translation environment need be preceded by only a decoder, since a decoder transforms data into a data set conforming to a memory-resident representation suitable for further processing by this translator.

A translator defined within the Miranda translation environment, however, requires a more complex combination of modules. This is because data must be transformed into a memory-resident representation using Miranda data structures before being transformed by a translator defined in the Miranda translation environment. A Miranda translator will therefore be preceded by the following sequence of transformations:

$$\mathcal{S} \overset{*}{\rightarrow} \mathcal{T}_C(C_S,\ I_S,\ P_C) \overset{encode}{\longrightarrow} \mathcal{T}_{mf}(C_S,\ I_S,\ P_{mf}) \overset{decode}{\longrightarrow} \mathcal{T}_{mm}(C_S,\ I_S,\ P_{mm})$$

where

$\mathcal{S}$ is the source data set to be transferred;

$\mathcal{T}_C(C_S,\ I_S,\ P_C) \overset{encode}{\longrightarrow} \mathcal{T}_{mf}(C_S,\ I_S,\ P_{mf})$
: is performed by an encoder to transform the values $\mathcal{T}_C$, having a memory-resident representation using C data structures, into the values $\mathcal{T}_{mf}$ having a text file representation using Miranda data structures;

$\mathcal{T}_{mf}(C_S,\ I_S,\ P_{mf}) \overset{decode}{\longrightarrow} \mathcal{T}_{mm}(C_S,\ I_S,\ P_{mm})$
: is performed by a decoder, implemented within the Miranda programming environment, to transform the data values $\mathcal{T}_{mf}$ into a set of values $\mathcal{T}_{mm}(C_S,\ I_S,\ P_{mm})$ having

a memory-resident representation using Miranda data structures suitable for the translator.

After the Miranda translator, the following sequence of transformations will typically occur:

$$\mathcal{T}_{mm'}(C_D,\ I_D,\ P_{mm'}) \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf'}(C_D,\ I_D,\ P_{mf'}) \stackrel{decode}{\longrightarrow} \mathcal{T}_{C'}(C_D,\ I_D,\ P_{C'}) \stackrel{*}{\rightarrow} \mathcal{D}$$

where

$\mathcal{T}_{mm'}(C_D,\ I_D,\ P_{mm'}) \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf'}(C_D,\ I_D,\ P_{mf'})$
: is performed by an encoder, implemented within the Miranda programming environment, to transform the values $\mathcal{T}_{mm'}$ into the values $\mathcal{T}_{mf'}$ having a text file representation using Miranda data structures;

$\mathcal{T}_{mf'}(C_D,\ I_D,\ P_{mf'}) \stackrel{decode}{\longrightarrow} \mathcal{T}_{C'}(C_D,\ I_D,\ P_{C'})$
: is performed by a decoder to transform the data values $\mathcal{T}_{mf'}$ into a set of values $\mathcal{T}_{C'}$ having a memory-resident representation using C data structures;

$\mathcal{D}$ is the transferred data set.

Inclusion of a Miranda translator within an interface performing the transfer $\mathcal{S} \stackrel{*}{\rightarrow} \mathcal{D}$ typically results in the following data transformations:

$$\mathcal{S} \stackrel{*}{\rightarrow} \mathcal{T}_C \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf} \stackrel{decode}{\longrightarrow} \mathcal{T}_{mm} \stackrel{translate}{\longrightarrow} \mathcal{T}_{mm'} \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf'} \stackrel{decode}{\longrightarrow} \mathcal{T}_{C'} \stackrel{*}{\rightarrow} \mathcal{D}.$$

Although use of the Miranda translation environment requires more data transformations than use of the C translation environment, the additional transformations $\mathcal{T}_C \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf}$, $\mathcal{T}_{mf'} \stackrel{decode}{\longrightarrow} \mathcal{T}_{C'}$, $\mathcal{T}_{mf} \stackrel{decode}{\longrightarrow} \mathcal{T}_{mm}$ and $\mathcal{T}_{mm'} \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf'}$ are performed by software that is constructed by the software tool `a2b`, as will be explained in Section 10.2. This requires no explicit definition within a transfer specification. Choosing between a C translation environment and a Miranda translation environment is therefore decided on the basis of efficiency and ease of use, rather than the number of transformations that have to be defined within the transfer specification. Choosing a translation environment is discussed further in Section 7.3.1.

6.2.2.1 Redundant encode and decode transformations

A transfer requiring one translation module which is defined within the C translation environment, has the following general form:

$$\mathcal{S} \stackrel{decode}{\longrightarrow} \mathcal{T}_0 \stackrel{translate}{\longrightarrow} \mathcal{T}_1 \stackrel{encode}{\longrightarrow} \mathcal{D}.$$

A transfer may, however, comprise two or more separate translation modules. For example, the data set $\mathcal{S}$ conforming to a source representation definition may be translated into the data set $\mathcal{I}$, conforming to a representation defined by a standard interchange format, before being translated into the data set $\mathcal{D}$ conforming to a destination representation. Such a transfer may be described as:

$$\mathcal{S} \stackrel{decode}{\longrightarrow} \mathcal{T}_0 \stackrel{translate}{\longrightarrow} \mathcal{T}_1 \stackrel{encode}{\longrightarrow} \mathcal{I} \stackrel{decode}{\longrightarrow} \mathcal{T}_2 \stackrel{translate}{\longrightarrow} \mathcal{T}_3 \stackrel{encode}{\longrightarrow} \mathcal{D}$$

where

- the two translation modules are defined within the C translation environment;
- the data sets $\mathcal{S}$ and $\mathcal{T}_0$ conform to the same conceptual and implementation schemas of the source representation, but $\mathcal{S}$ has a text file representation, and $\mathcal{T}_0$ has a memory-resident representation;
- the data sets $\mathcal{T}_1$, $\mathcal{I}$, and $\mathcal{T}_2$ conform to the same conceptual and implementation schemas of the representation defined for the interchange format. However, the data sets $\mathcal{T}_1$ and $\mathcal{T}_2$ have the same memory-resident representation, and the data set $\mathcal{I}$ has a text file representation;
- the data sets $\mathcal{T}_3$ and $\mathcal{D}$ conform to the conceptual and implementation schemas of the destination representation, but $\mathcal{T}_3$ has a memory-resident representation, and $\mathcal{D}$ has a text file representation.

The two data sets $\mathcal{T}_1$ and $\mathcal{T}_2$ are identical, therefore the transformations $\mathcal{T}_1 \stackrel{encode}{\longrightarrow} \mathcal{I} \stackrel{decode}{\longrightarrow} \mathcal{T}_2$ are redundant and the optimal sequence of transformations for this transfer would be:

$$\mathcal{S} \stackrel{decode}{\longrightarrow} \mathcal{T}_0 \stackrel{translate}{\longrightarrow} \mathcal{T}_1 \stackrel{translate}{\longrightarrow} \mathcal{T}_3 \stackrel{encode}{\longrightarrow} \mathcal{D}.$$

When consecutive translator modules are defined within the Miranda translation environment, some of the additional encode and decode transformations discussed in the introduction to Section 6.2.2 are also redundant for the same reasons that the transformations $\mathcal{T}_1 \stackrel{*}{\rightarrow} \mathcal{T}_2$ are redundant. The sequence of transformations performed between consecutive Miranda translator modules may be described, using subscripts similar to those in the introduction to Section 6.2.2, as follows:

$$\begin{aligned}\mathcal{S} \stackrel{*}{\rightarrow} \mathcal{T}_{mm} &\stackrel{translate}{\longrightarrow} \mathcal{T}_{mm'} \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf'} \stackrel{decode}{\longrightarrow} \mathcal{T}_{C'} \stackrel{encode}{\longrightarrow} \mathcal{I} \\ &\stackrel{decode}{\longrightarrow} \mathcal{T}_{C''} \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf''} \stackrel{decode}{\longrightarrow} \mathcal{T}_{mm''} \stackrel{translate}{\longrightarrow} \mathcal{T}_{mm'''} \stackrel{*}{\rightarrow} \mathcal{D}.\end{aligned}$$

where the data sets $\mathcal{T}_{mm'}$ and $\mathcal{T}_{mm''}$ are identical. Removing redundant encode and decode transformations, the optimised sequence of transformations is:

$$\mathcal{S} \xrightarrow{*} \mathcal{T}_{mm} \overset{translate}{\longrightarrow} \mathcal{T}_{mm'} \overset{translate}{\longrightarrow} \mathcal{T}_{mm''} \xrightarrow{*} \mathcal{D}.$$

Section 8.3.2 contains a description of the technique developed by the author for producing an optimal sequence of transformations for a combination of interface modules.

6.3 An example

To conclude this Section, an example is given of a combination of interface modules that perform a transfer having the general form defined by the interchange format interfacing strategy. This transfer also includes the movement of the data from the source computer system to the destination computer system through a communication network and is described in detail as:

Application	$\mathcal{S}$	$\overset{decode}{\longrightarrow}$	$\mathcal{T}_0$	$\overset{translate}{\longrightarrow}$	$\mathcal{T}_1$	$\mathcal{T}_2$	$\overset{translate}{\longrightarrow}$	$\mathcal{T}_3$	$\overset{encode}{\longrightarrow}$	$\mathcal{D}$
Presentation			$\mathcal{T}_1$	$\overset{encode}{\longrightarrow}$	$\mathcal{N}_S$	$\mathcal{N}_D$	$\overset{decode}{\longrightarrow}$	$\mathcal{T}_2$		
Layers 1–5					$\mathcal{N}_S$	$\overset{communicate}{\rightsquigarrow}$	$\mathcal{N}_D$			

The modules to perform this transfer are:

- a decoder and a C translator to perform application layer transformations that modify data values to conform to the conceptual and implementation schemas of the interchange format. These transformations are described as:

$$\mathcal{S}(C_S,\ I_S,\ P_S,\ L_S) \overset{decode}{\longrightarrow} \mathcal{T}_0(C_S,\ I_S,\ P_0,\ L_S) \overset{translate}{\longrightarrow} \mathcal{T}_1(C_I,\ I_I,\ P_1,\ L_S)$$

- an encoder to perform the presentation layer transformation that modifies the interchange data set $\mathcal{T}_1$ to conform to the physical schema of the network representation. This transformation is described as:

$$\mathcal{T}_1(C_I,\ I_I,\ P_1,\ L_S) \overset{encode}{\longrightarrow} \mathcal{N}_S(C_I,\ I_I,\ P_N,\ L_S)$$

- a communicator to perform the transformations occurring at the 5 lower levels of the ISO-OSI Reference Model. These transformations move the data set $\mathcal{N}_S$ from one computer system to another, and are collectively described as:

$$\mathcal{N}_S(C_I,\ I_I,\ P_N,\ L_S) \overset{communicate}{\rightsquigarrow} \mathcal{N}_D(C_I,\ I_I,\ P_N,\ L_D)$$

- a decoder to perform the presentation layer transformation that modifies the data set $\mathcal{N}_D$ to conform to a temporary physical schema. This transformation is described as:

$$\mathcal{N}_D(C_I,\ I_I,\ P_N,\ L_D) \xrightarrow{decode} \mathcal{T}_2(C_I,\ I_I,\ P_2,\ L_D)$$

- a C translator and an encoder to perform application layer transformations for modifying the interchange data set $\mathcal{T}_2$ into a data set conforming to the conceptual, implementation and physical schemas used for the destination geographical information system. These transformations are described as:

$$\mathcal{T}_2(C_I,\ I_I,\ P_2,\ L_D) \xrightarrow{translate} \mathcal{T}_3(C_D,\ I_D,\ P_3,\ L_D) \xrightarrow{encode} \mathcal{D}(C_D,\ I_D,\ P_D)$$

In the next Chapter, a description is given of how to specify a transfer in a way that allows an interface and its constituent modules to be generated from this specification.

Transfer specifications

Chapter 7

An interface is constructed according to a *transfer specification* which comprises:

an interface definition
: specifying the combination of modules that will form the interface to perform the transfer;

two or more representation definitions
: each specifying a collection of data types, and a text file representation for values of these types; and

one or more translation definitions
: each specifying the translation of data values from one representation to another.

Encoder, and decoder modules are generated from representation definitions, and translator modules are constructed using translation definitions. Although communication modules have a fixed form, to be described in Chapter 11, the representation of any data values to be sent through a communications network by these modules is defined in part by a representation definition, and in part by the communication standard ISO 8825 (ISO8825 1987), to be discussed in Appendix A.3.

An interface definition is expressed using the notation to be defined in Section 7.1. A representation definition is expressed using the notation called A2B, developed by the author and to be defined in Section 7.2. As will be explained in Section 7.3, translations are defined using the constructs and operations provided by either the Miranda functional language, or the C programming language. A future goal of the author, to be discussed further in Section 14.4, is to define a translation environment comprising a notation for specifying the translation of data values from one representation to another in terms of the two representation definitions expressed using A2B.

Before discussing the various definitions that form a transfer specification, an example is now introduced which will be used extensively throughout the remainder of this thesis. The source data representation is a simplified form of the GeoVision Gina format (GeoVision 1986), and the destination data representation is similar to the text file representation of vector data files used for the GRASS geographical information system (Shapiro, Westervelt, Gerdes, Larsen & Brownfield 1992). Transfer of the source data set $\mathcal{S}$ into the destination data set $\mathcal{D}$ is described as:

$$\mathcal{S} \overset{decode}{\longrightarrow} \mathcal{T}_0 \overset{translate}{\longrightarrow} \mathcal{T}_1 \overset{encode}{\longrightarrow} \mathcal{D}.$$

Examples of the data sets $\mathcal{S}$ and $\mathcal{D}$ are given in Figure 7.1.

```
feat 874 L
coor 867 543 874 550 880 564
feat 875 L
coor 880 564 876 580
feat 876 P
coor 872 550
```

The source data set $\mathcal{S}$

```
Line 874 3 867 543 880 564
867 543
874 550
880 564
Line 875 2 876 564 880 580
880 564
876 580
Point 876 872 550
```

The destination data set $\mathcal{D}$

Figure 7.1: Example data sets

For comparison, this transfer is the same as the one used in Section 5.2.4 to demonstrate the author's earlier work. In Figure 7.2, a partial specification of this transfer is presented using the notation to be defined in this Chapter. Two complete transfer specifications are presented in Appendix B.

The transfer specification in Figure 7.2 comprises:

- an interface definition expressed using the notation to be defined in Section 7.1;
- two representation definitions expressed using the notation to be described in Section 7.2; and
- a partial translation definition. Definition of translators is discussed in Section 7.3, where two complete translation definitions for this transfer specification are given in Figures 7.9 and 7.10.

Throughout this Chapter, BNF is used to define a notation for specifying a transfer. Unfortunately BNF and the notation being defined have some symbols in common, for example

Interface definition

```
{ %a2b source to destination
```

Representation definition

```
%source
source:      (feature || "\n" )* ;
feature:     ("feat" (id type) "\ncoor" (point)*);
point:       (x y);
id, x, y:    integer;
type:        << "L" | "P" >>;
```

Representation definition

```
%destination
destination: (< "Line " ((id #nPts minBoundRect) "\n"
                              (point || "\n")^nPts)
               | "Point " ( id x y )> "\n")*;
point:       (x y);
minBoundRect:(mnx mny mxx mxy);
id, nPts, x, y, mnx, mny, mxx, mxy:integer;
```

Translation definition

```
%source2destination
% .. {
..
%}
```

Figure 7.2: Transfer specification for the example of Figure 7.1

brackets and braces. To avoid confusion any BNF description is written here without using the symbols { }, or ( ). A BNF description such as

$$\langle \text{identifier} \rangle \longrightarrow \langle \text{letter} \rangle \ \{\langle \text{letter} \rangle \mid \langle \text{digit} \rangle\}$$

would be written here as follows:

$$\begin{aligned} \langle \text{identifier} \rangle &\longrightarrow \langle \text{letter} \rangle \\ &\mid \langle \text{letter} \rangle \langle \text{letters and digits} \rangle \\ \langle \text{letters and digits} \rangle &\longrightarrow \langle \text{letter} \rangle \langle \text{letters and digits} \rangle \\ &\mid \langle \text{digit} \rangle \langle \text{letters and digits} \rangle \end{aligned}$$

The symbol | is also common to both. Therefore, when using this symbol to indicate alternative BNF descriptions this symbol is placed below the BNF symbol $\longrightarrow$, as shown above. Used anywhere else, this symbol is a part of the notation being defined. The symbols $\langle$ $\rangle$, and [ ] are a part of the BNF notation and are distinct from the symbols < >, and [ ] which are a part of the notation being defined.

In the following Sections, an explanation is given of the notation used for specifying a transfer. Differences exist between the notation defined here, and the notation processed by the software tool `a2b`. These differences, to be described in Appendix D, are a consequence of the notation evolving in light of the author's experience with using `a2b`.

7.1 Interface definition

An interface definition is expressed using a notation that is formally specified by the following BNF grammar:

⟨interface definition⟩ ⟶ `%a2b` ⟨module combination⟩

⟨module combination⟩ ⟶ ⟨representation name⟩ ⟨connector⟩ ⟨representation name⟩

| ⟨interface definition⟩ ⟨connector⟩ ⟨representation name⟩

⟨connector⟩ ⟶ `to`

| `via` ⟨representation name⟩ `to`

Any interface definition has one of two general forms depending on whether a communicator module is included within the definition. Without any communicator modules, the general form of an interface definition is:

```
%a2b  repNameA to repNameB
```

where `repNameA` and `repNameB` are the names of two representation definitions included within the transfer specification. Transfers performed by interfaces having this general form are described as:

$$\mathcal{S} \xrightarrow{*} \mathcal{D} \tag{7.1}$$

With a communicator module, the general form of an interface definition is:

```
%a2b repNameA via comName to repNameB
```

where `repNameA`, `comName`, and `repNameB` are the names of representation definitions given within the transfer specification. Any name of a representation definition preceded by the keyword `via` is a reference to a representation definition which specifies the types of values to be sent through a network by a communicator module. Transfers performed by interfaces having this general form are described as:

$$\mathcal{S} \xrightarrow{*} \mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D \xrightarrow{*} \mathcal{D} \tag{7.2}$$

where

- the data set $\mathcal{S}$ conforms to the representation defined by `repNameA`;
- the data sets $\mathcal{N}_S$ and $\mathcal{N}_D$ conform to the representation defined by `comName`, but are located on different computer systems; and
- the data set $\mathcal{D}$ conforms to the representation defined by `repNameB`.

The two forms (7.1) and (7.2) of an interface definition may be combined to form a variety of interface definitions. For example:

```
%a2b gina via macdif to grass to dlg
```

defines an interface which performs a transfer described as:

$$\mathcal{S}_{gina} \xrightarrow{*} \mathcal{N}_{macdif} \overset{communicate}{\rightsquigarrow} \mathcal{N}_{macdif'} \xrightarrow{*} \mathcal{T}_{grass} \xrightarrow{*} \mathcal{D}_{dlg}.$$

This interface transforms data values $\mathcal{S}_{gina}$ conforming to the Gina representation into the data set $\mathcal{N}_{macdif}$ and moves this set through a network to become the data set $\mathcal{N}_{macdif'}$. Both $\mathcal{N}_{macdif}$ and $\mathcal{N}_{macdif'}$ conform to the MACDIF representation, but are located on different computer systems. The data set $\mathcal{N}_{macdif'}$ is transformed into data sets conforming to the GRASS, and then finally the DLG format. Gina (GeoVision 1986) is the text file format defined for the GeoVision geographical information system; MACDIF was discussed in Section 4.4; GRASS (Shapiro *et al* 1992), is a public domain geographical information system; and DLG (Geological Survey 1990) is the Digital Line Graph format.

Translation modules are implied within any interface definition. The implied translators 'occur' between any two successive references to different representation definitions. Thus, the interface definition

```
%a2b source to destination
```

requires a translator to be generated from a translation definition provided by the person constructing the interface. The definition of an implied translator has a name comprising the names of the two representation definitions, separated by the character '2'. In the above example, the module name of the translation definition is `source2destination`. Similarly, the interface definition

```
%a2b source via netrep to destination
```

requires two translator definitions called `source2netrep` and `netrep2destination`, and defines an interface which performs a transfer described as:

$$\mathcal{S}_{source} \xrightarrow{*} \mathcal{N}_{netrep} \rightsquigarrow \mathcal{N}_{netrep'} \xrightarrow{*} \mathcal{D}_{destination}.$$

7.2 Representation definitions

In this Section, a notation is defined for specifying the representation of various types of data values. An early version of this notation called CAS (Concrete and Abstract Syntax) is described in (Pascoe & Penny 1993); however, this notation has been revised to become the notation A2B described in this Section. Although intended primarily for defining a text file representation, any representation definition will also determine a memory-resident representation using C data structures (to be discussed in Section 9.1.3), and a network representation (to be discussed in Chapter 11).

A BNF description of a representation definition is given in Figure 7.3.

⟨representation definition⟩	⟶	% ⟨representation name⟩ ⟨options⟩ ⟨rules⟩
⟨options⟩	⟶	ε
	|	⟨option⟩ ⟨options⟩
⟨rules⟩	⟶	⟨rule⟩
	|	⟨rule⟩ ⟨rules⟩

Figure 7.3: Structure of a representation definition

7.2.1 Definition of data types

Representation of various types of data values is specified in the form of rules. The structure of a rule is defined using BNF in Figure 7.4.

Any representation definition comprises at least one rule which has a name the same as that of the representation definition. In Figure 7.2, for example, definition of the representation called `source` is started by a rule which defines a type named `source`, and definition of the representation called `destination` is started by a rule which defines a type named `destination`.

Definition of data types using rules is based on the type constructs specified for Abstract

⟨rule⟩ ⟶ ⟨type names⟩ : ⟨type⟩ ;
⟨type names⟩ ⟶ ⟨type name⟩
| ⟨type name⟩ , ⟨type names⟩
⟨type name⟩ ⟶ ⟨letter⟩
| ⟨type name⟩ ⟨letter or digit⟩
⟨type⟩ ⟶ ⟨type definition⟩
| ⟨type name⟩
⟨type definition⟩ ⟶ ⟨literals⟩ ⟨simple type definition⟩ ⟨literals⟩
| ⟨literals⟩ ⟨structured type definition⟩ ⟨literals⟩

Figure 7.4: Structure of rules within representation definitions

Syntax Notation.1 (ASN.1) (ISO8824 1987), to be described in Appendix A.2. In the definition of ASN.1, a *simple type* is 'a type defined by directly specifying the set of its values', a *structured type* is 'a type defined by a reference to one or more other types', and a *component type* is 'one of the types referenced when defining a structured type'.

Literals are used to specify the text file representation of values of the various types specified within a representation definition, and are discussed in Section 7.2.2. They do not form any part of the type definition, only the way in which values of this type are to be represented in a text file. The literals "`feat`", "`coor`", "`Line`", and "`Point`" in Figure 7.1, for example, distinguish between the different text file records in which data values are represented.

7.2.1.1 Simple type definitions

The notation for defining the representation of simple types is described using BNF in Figure 7.5. The two non-terminals ⟨regular expression⟩ and ⟨encoding control string⟩ used in this BNF description are explained in Section 7.2.2.1.

The notation defined here for specifying a representation makes use of four simple types:

integer

comprising data values which are the positive and negative whole numbers;

real

comprising data values which are the positive and negative numbers which may have a fractional part;

string

comprising data values which are any sequence of alphanumeric characters; and

⟨simple type definition⟩ ⟶ ⟨simple type keyword⟩
| ⟨simple type keyword⟩ ⟨text file representation⟩
| ⟨enumerated type⟩
⟨simple type keyword⟩ ⟶ `integer`
| `real`
| `string`
⟨text file representation⟩ ⟶ : ⟨integer expression⟩
| = (⟨regular expression⟩ : ⟨encoding control string⟩)
⟨enumerated type⟩ ⟶ << ⟨enumerated values⟩ >>
⟨enumerated values⟩ ⟶ " ⟨character sequence⟩ "
| " ⟨character sequence⟩ " | ⟨enumerated values⟩
⟨integer expression⟩ ⟶ ⟨integer value⟩
| ⟨integer type name⟩
| ⟨integer expression⟩ + ⟨integer expression⟩
| ⟨integer expression⟩ - ⟨integer expression⟩
| ⟨integer expression⟩ / ⟨integer expression⟩
| ⟨integer expression⟩ * ⟨integer expression⟩
⟨integer value⟩ ⟶ ⟨digit⟩
| ⟨integer value⟩ ⟨digit⟩
⟨integer type name⟩ ⟶ ⟨type name⟩

Figure 7.5: BNF description of the notation for defining simple data types

enumerated

comprising data values explicitly defined by quoted sequences of characters within the type definition.

Use of the integer, real, and string simple types to define a new type is indicated by the keywords `integer`, `real`, and `string`. The `source` representation defined in Figure 7.2, for example, contains a definition of the integer simple types called `id`, `x`, and `y`. An enumerated type is defined by listing all the values comprising the data type. For example, in Figure 7.2 the type named `type` defined in the `source` representation definition is an enumerated type comprising the two values `"L"` and `"P"`.

Simple types are combined to form structured data types, which are based on the notions of sets and sequences of values, and on choices among different data types. The form of a structured type definition is described using BNF in Figure 7.6. Sequence type definitions are described in Section 7.2.1.2, set type definitions are described in Section 7.2.1.3, and choice type definitions are described in Section 7.2.1.4.

⟨structured type definition⟩ ⟶ ⟨set type definition⟩
| ⟨sequence type definition⟩
| ⟨choice type definition⟩
⟨set type definition⟩ ⟶ ⟨set⟩
| ⟨set-of⟩
| ⟨set-plus⟩
| ⟨N-set⟩
⟨set⟩ ⟶ { ⟨component types⟩ }
⟨set-of⟩ ⟶ { ⟨component type⟩ ⟨separator⟩ }*
⟨set-plus⟩ ⟶ { ⟨component type⟩ ⟨separator⟩ }+
⟨N-set⟩ ⟶ { ⟨component type⟩ ⟨separator⟩ }^ ⟨integer expression⟩
⟨sequence type definition⟩ ⟶ ⟨sequence⟩
| ⟨sequence-of⟩
| ⟨sequence-plus⟩
| ⟨N-sequence⟩
⟨sequence⟩ ⟶ (⟨component types⟩)
⟨sequence-of⟩ ⟶ (⟨component type⟩ ⟨separator⟩)*
⟨sequence-plus⟩ ⟶ (⟨component type⟩ ⟨separator⟩)+
⟨N-sequence⟩ ⟶ (⟨component type⟩ ⟨separator⟩)^ ⟨integer expression⟩
⟨choice type definition⟩ ⟶ < ⟨choice component types⟩ >
⟨choice component types⟩ ⟶ ⟨type reference⟩
| ⟨type reference⟩ | ⟨choice component types⟩
⟨separator⟩ ⟶ ε
| | | ⟨literal⟩
⟨component types⟩ ⟶ ⟨component type⟩
| ⟨component types⟩ [⟨literals⟩ ⟨component types⟩ ⟨literals⟩]
| ⟨component types⟩ ⟨literals⟩ ⟨component type⟩
⟨component type⟩ ⟶ ⟨type reference⟩
| ⟨type definition⟩
⟨type reference⟩ ⟶ ⟨integer type name⟩
| # ⟨integer type name⟩
| ⟨type name⟩

Figure 7.6: BNF description of the notation for defining structured data types

7.2.1.2 Sequences

The notation defined here for specifying a representation comprises four kinds of structured data types based on the notion of a sequence. These four sequence data types are:

sequence

'defined by referencing a fixed, ordered, list of types (some of which may be declared to be optional); each value of the new type is an ordered list of values, one from each component type' (ISO8824 1987);

sequence-of

'defined by referencing a single existing type; each value in the new type is an ordered list of zero, one or more values of the existing type' (ISO8824 1987);

sequence-plus

which is a sequence-of type where a value of the new type is an ordered list of *one or more* values of the existing type; and

N-sequence

which is also a sequence-of type where a value of the new type is an ordered list of some *fixed number* of values of the existing type.

Sequences and sets (discussed in the next Section) comprise a fixed list of component types. A component type may, however, be optional, and such a component type is enclosed within the symbols '`[`' and '`]`'. Any component type may be either a type reference, or a type definition.

Component type references are the names of types defined in the representation definition, with no type being referenced more than once within any list of component types. Integer type references may be preceded by the symbol '`#`' to indicate that values of this type are to be used in the definition of other data types specified within the representation definition. An example is shown in the definition of the `destination` representation where the integer type reference `nPnts` is preceded by the symbol '`#`', and then used later to determine the number of point values comprising an N-sequence.

Component type definitions have no type name and are referred to here as anonymous type definitions. Only one of each kind of simple type definition is allowed within any list of component types. Multiple references to any one kind of simple type is accomplished by defining new types of this kind. An example of this technique is given in the definition of the `source` representation in Figure 7.2, where the two integer types `x`, and `y` are defined and referenced as components of the `point` structured type definition.

Various sequence data types are defined in the `source` representation definition given in Figure 7.2. The type named `source`, for example, is defined to be a sequence of zero, one, or more values of the type named `feature`. Each value of the type named `feature` is defined to comprise a sequence of two values, the types of which are unnamed. The first of these is a sequence value comprising two components, the integer simple type called `id`, and an enumerated sequence type called `type`. The second of these values is a sequence of zero, one, or more values of the type named `point`.

7.2.1.3 Sets

The notation defined here for specifying a representation comprises four kinds of structured data types based on the notion of a set. These four set data types are:

set

'defined by referencing a fixed, unordered, list of distinct types (some of which may be declared to be optional); each value in the new type is an unordered list of values, one from each of the component types' (ISO8824 1987);

set-of

'defined by referencing a single existing type; each value in the new type is an unordered list of zero, one or more values of the existing type' (ISO8824 1987);

set-plus

which is a set-of type where a value of the new type is an unordered list of *one or more* values of the existing type; and

N-set

which is a set-of type where a value of the new type is an unordered list of some *fixed number* of values of the existing type.

Rules governing the use and definition of the various set data types are the same as for the sequence data types discussed in the preceding Section. The only difference between the two is that component types of a set data type are unordered, whereas component types of a sequence data type are to occur in the order they are specified in the type definition.

7.2.1.4 Choices

A choice type is defined by 'referencing a fixed, unordered, list of distinct types; each value of the new type is one of the component types' (ISO8824 1987). An example of defining a choice type is given in Figure 7.2 on page 99. The new type is called '`destination`' and values of this new type will be one of two unnamed types:

- a sequence comprising two elements. One, a sequence consisting of three component types 'id', 'nPnts', and 'minBoundRect'. The other, an N-sequence comprising a fixed number of 'point' values according to the value of the preceding 'nPts' type; and
- a sequence comprising the three integer simple types called 'id', 'x', and 'y'.

Methods for specifying the text file representation of data within representation definitions is described below.

7.2.2 Definition of text file representations

Defining text file representations for values of the various data types specified within a representation definition is described in three parts. Defining the text file representation of simple types is discussed in Section 7.2.2.1 and the text file representation of structured types is discussed in Section 7.2.2.2. Separating the text file representation of values using delimiters such as space, tab, and newline is referred to here as *delimiter insertion*, and is discussed in Section 7.2.2.3.

7.2.2.1 Simple types

When defining the values of an enumerated type, the sequence of characters that define any value is the text file representation of that value. Text file representations for integer, string, or real values are specified by regular expressions for defining the method of decoding values, and encoding control strings for defining the method of encoding values. Default text file representations for the integer, string, and real types of values are shown in Table 7.1.

Simple type of value	Regular expression	Encoding control string
integer	[+-]?[0-9]+	"%d"
real	[+-]?[0-9]*\.[0-9]+([Ee][-+][0-9][0-9])?	"%f"
string	\"((\\.)|[^\"\\])*\"	"%s"

Table 7.1: Text file representation for values of simple types

As will be explained in Chapter 9, regular expressions are processed by the software tool `flex` (Paxson 1990) to generate a scanner for reading values which are represented within a text file. These regular expressions must therefore conform to the requirements of `flex`. Encoding control strings are used by the `printf` function, provided in the standard input/output library for the C programming language, to encode values within a text file. Therefore, encoding control strings must conform to that required for the function `printf`.

Different text file representations may be defined for integer, real and string simple types within a representation according to the ⟨text file representation⟩ BNF production shown in Figure 7.5. An example of specifying a different text file representation for a string simple type is as follows:

```
quote : string = ( "'([A-Za-z ])*'" : "'%s'");
```

with a value of this type being: `'An example quote value'`.

In some geographical data file formats such as Colourmap (CSIRONET 1986), values of simple types are represented using a fixed number of characters. For example, 'All integer values must be right justified in the 10 byte fields' (CSIRONET 1986, page 90). Simple types comprising values which are represented by a fixed number of characters may also be specified using the notation defined here. For example, an integer type for the Colourmap data file format is specified as follows:

```
colourmapInteger : integer:10;
```

In other file formats, the number of characters used to represent a value is given within the data file. Consider the Spatial Data Transfer Standard (Geological Survey 1992), for example, which uses the text file representation defined by the standard ISO 8211 (ISO8211 1985), discussed in Appendix A.1. The data descriptive record contains a directory entry map which defines the number of characters used to represent each of the tag, length, and position fields of an entry in the directory. Using the notation defined in this Chapter, this part of an SDTS representation is defined as:

```
entryMap:           ( #sizeFieldLength #sizeFieldPosition
                      reserved1 #sizeFieldTag );
directoryEntry:     ( fieldTag fieldLength fieldPosition );
fieldTag:           string:sizeFieldTag;
fieldLength:        integer:sizeFieldLength;
fieldPosition:      integer:sizeFieldPosition;
sizeFieldPosition,
sizeFieldLength,
sizeFieldTag:       integer:1;
reserved1:          string:1;
```

Values of the `sizeFieldLength`, `sizeFieldPosition`, and `sizeFieldTag` types determine the number of characters used to represent values of the types `fieldPosition`, `fieldLength`, and `fieldTag` respectively. Defining an SDTS representation is to be discussed further in Chapter 13.

7.2.2.2 Structured types

Text file representations for values of structured data types are defined by the text file representation of the component types, and by the use of literals within the structured type definition.

Specifying the content and position of literals is central to defining the text file representation of structured data values. The structure of a literal is described using BNF in Figure 7.7. Essentially, a literal comprises a sequence of characters enclosed within quotes.

⟨literal⟩ ⟶ " ⟨character sequence⟩ "
⟨character sequence⟩ ⟶ ε
| ⟨alphanumeric character⟩ ⟨character sequence⟩
| ⟨escape character⟩ ⟨character sequence⟩
⟨literals⟩ ⟶ ε
| ⟨literal⟩ ⟨literals⟩

Figure 7.7: Structure of literals

The escape characters shown in Table 7.2 may be embedded within a character sequence to represent certain non-graphical characters. More generally, the backslash character \ may

Escape code	Character
`\a`	alert (bell)
`\b`	backspace
`\f`	form feed
`\n`	newline
`\r`	carriage return
`\t`	horizontal tab
`\v`	vertical tab

Table 7.2: Escape characters within literal definitions

be used to include any awkward character such as a double quote or a backslash within the literal. For example, the literal definition for the character sequence :"\": is ":\"\\\":".

Literals may be inserted before or after any type specified in a rule. Literals may also be inserted between component types of sets and sequences, and the various data types that form a choice data type. Pairs of values within a set-of, set-plus, N-set, sequence-of, sequence-plus, or N-sequence value may be separated by a literal that is specified in a separator as part of the type definition. The `source` data type defined within the `source` representation

in Figure 7.2, for example, comprises a sequence of `feature` values, with each pair of feature values in this sequence separated by a newline character.

Data values to be encoded beside each other may need to be separated by a delimiter to avoid the two values being misinterpreted as one value. Delimiter insertion is described next.

7.2.2.3 Delimiter insertion

Delimiter insertion is the process of inserting either a space, tab, or newline character, or some other delimiter specified within the representation definition between the text file representations of two data values. For example, consider the `point` type defined in both the representation definitions shown in Figure 7.2. Given an `x` value of `34`, a `y` value of `61`, and no use of delimiters, this point value would be encoded into the following text file representation:

```
3461
```

The two integer values `34` and `61` could easily be mistaken as being the single integer value `3461`. Inserting the delimiter ",", the `point` value would have the following text file representation:

```
34,61
```

where the two values are easily recognised.

In some text file representations such as the Colourmap format (CSIRONET 1986), for example, values of the `x` and `y` data types may be represented by a fixed number of characters. Thus, the `point` type definition would be:

```
point : ( x y );
x,y : integer:10;
```

where the notation `integer:10` would indicate that integer values of the data types `x` and `y` are each represented by 10 characters. Data values represented using a fixed number of characters require no delimiters, and the two integers of the example point value given above would be represented by the 20 characters

␣␣␣␣␣␣␣␣34␣␣␣␣␣␣␣␣61

where the symbol ␣ represents one space. Note that a single integer value may occupy the entire 10 characters such that the sequence ␣␣␣8674531997826510 represents two integer values `8674531` and `9978265104`.

Delimiter insertion may be explicitly specified within the options part of a representation definition according to the BNF grammar shown in Figure 7.8.

As shown in Figure 7.8, explicit delimiter insertion may be specified in one of two ways:

```
⟨delimiter insertion options⟩ ⟶ @dfspace ⟨delimiter⟩
                              |  @space ⟨data type⟩ ⟨data type⟩ ⟨delimiter⟩
          ⟨data type⟩         ⟶ ⟨simple type keyword⟩
                              |  ⟨literal⟩
                              |  *
```

Figure 7.8: Notation for specifying explicit delimiter insertion

`@dfspace`⟨delimiter⟩

which specifies the default delimiter to insert between any two text file representations of values that *need* to be separated; and

`@space`⟨data type⟩ ⟨data type⟩ ⟨delimiter⟩

which specifies a delimiter to insert between any pair of values of the specified simple types, literals, or some combination of the two. Use of the symbol * as a type name indicates that a value of any data type must be separated from the other data type by the given delimiter.

Currently, explicit delimiter insertion specifications affect only the encoding of values within a text file and have no influence over the decoding of values within a text file. More generally, the concept of delimiter insertion requires further development to establish the value of this approach to solving an awkward problem.

7.3 Translation definitions

Translation definitions are expressed using the constructs and operations provided by some *translation environment*. Examples reviewed in Chapter 5 included:

- a translation environment based upon hierarchical structures and operations provided by the languages DEFINE and CONVERT which were developed for EXPRESS (Section 5.1.1);
- a translation environment based upon using set notation to specify data transformations to be implemented as a Prolog program (Section 5.1.2); and
- translation environments based upon the structures and operations provided by the relational data model (Section 5.2.3).

A translation environment has, for the reasons given earlier in Section 3.4.1, generally been chosen to simplify the definition of the required data transformations rather than to construct efficient interfaces. Constructing efficient interfaces is, however, very important if data is to be freely exchanged across a network. In particular, efficient interfaces are essential to allow databases to be shared through a network.

In the approach to interface construction described in this thesis, data translations cannot be specified in terms of the set and sequence constructs used for defining representations. Instead, `a2b` has been developed to support two 'lower-level' translation environments:

1. the environment provided by the C programming language (Kernighan & Ritchie 1978), to allow efficient data translations to be constructed; and
2. the environment provided by the Miranda programming language (Turner 1986), to assess whether specifying data translations can be simplified by using a functional language.

These translation environments are discussed further in Section 7.3.1.

Ultimately, the author would like to provide an *a2b translation environment* comprising a notation that allows translations to be defined in terms of the representation definitions to which the data values conform before and after translation. Experience with EXPRESS, described earlier in Section 5.1.1, led Taylor (1982) to conclude that 'a data description capability coupled with a data manipulation capability yields a much more powerful facility than simply a data description capability by itself'. Possibilities for developing the `a2b` translation environment are discussed further in Section 14.4.

7.3.1 Translation environments supported by `a2b`

As a step towards developing the `a2b` translation environment, a procedural programming language and a functional programming language were chosen as the basis for two different translation environments:

the C translation environment
: in which translation definitions are specified using the C programming language; and

the Miranda translation environment
: in which translation definitions are specified using the Miranda programming language.

A translation definition comprises one or more functions expressed using either the C programming language, or the Miranda programming language. At least one function within any translation definition must have a particular form which is now described for the C and Miranda translation environments.

7.3.1.1 C Translation definitions

Any translation definition using the C programming language comprises at least one function which has the following general form:

```
repA2repB(varAdata, varBdatap)
    struct type_REPA_RepA  *varAdata;
    struct type_REPB_RepB **varBdatap;
{
...
}
```

The name of this function, `repA2repB`, comprises the concatenation of: the name of the representation definition (`repA`) to which the data conforms before translation; the symbol '2'; and the name of the representation definition (`repB`) to which the data conforms after translation. The C function `repA2repB` transforms the data set pointed to by the variable called `varAdata` into another data set that is indirectly pointed to by the variable called `varBdatap`. This second level of indirection is necessary for the transformed data values to be returned to the main routine of the interface.

The variable `varAdata` is a pointer to a C data structure. The definition of this C data structure is generated by the software tool `pepsy`, as will be described in Chapter 9. The name of this C data structure comprises the concatenation of: the symbol '`type_`'; the name, in uppercase, of the representation definition from which the structure declaration was generated; the symbol '`_`'; and the name of the representation definition once again, this time with only the first character in uppercase and the remainder as given in the transfer specification. The type name of the C structure indirectly pointed to by the variable `varBdata` is constructed in the same way.

Data conforming to the `source` representation definition given in Figure 7.1 is transformed by a translator module generated from the C translation definition, given in Figure 7.9, into another data set conforming to the `destination` representation definition also given in Figure 7.1. The C translation function given in Figure 7.9 is therefore called `source2destination`. The variable `s` points to a C data structure of the type called `type_SOURCE_Source`. The variable `dp` points to a pointer which in turn points to a C data structure of the type called `type_DESTINATION_Destination`.

7.3.1.2 Miranda translation definitions

Any translation definition using the Miranda programming language comprises at least one function which has the following general form:

```
%source2destination %c{
#include "define.h"
source2destination(s,dp)
    type_SOURCE_Source, *s;
    type_DESTINATION_Destination, **dp;
{
     Ddecl(Destination, *d); Sdecl(CasS2, *sP); Ddecl(CasS2, *dP);
     new(&d,Destination);*dp = d;
     for (;s;s=s->next,d=d->next) {
           new(&Dfeature,CasS8);
           switch (Sfeature->casS0->type) {
              case 0 : /* Line */
                 int init = 1;
                 Dfeature->offset = type_DESTINATION_CasS8_casS4;
                 new(&Dline,CasS4); new(&DlineInfo,CasS0);
                 DlineInfo->id = Sfeature->casS0->id;
                 DlineInfo->nPts = 0; DlineInfo->elm0 = NULL;
                 if (Sfeature->casS2) {
                      new(&Dline->casS2,CasS2); new(&DlineInfo->elm0,MinBoundRect);
                      for (sP = Sfeature->casS2, dP = Dline->casS2; sP;
                           sP = sP->next, dP=dP->next,init = 0) {
                             DlineInfo->nPts++;
                             new(&dP->member_DESTINATION_0,Point);
                             dP->member_DESTINATION_0->x = sP->element_SOURCE_1->x;
                             dP->member_DESTINATION_0->y = sP->element_SOURCE_1->y;
                             setMinMbr(mnx,x); setMinMbr(mny,y);
                             setMaxMbr(mxx,x); setMaxMbr(mxy,y);
                             setNext(sP->next,&dP->next,CasS2); } }
                     break;
               case 1 : /* Point */
                  Dfeature->offset = type_DESTINATION_CasS8_casS6;
                  new(&Dpoint,CasS6);
                  Dpoint->id = Sfeature->casS0->id;
                  Dpoint->x = Spoint->x; Dpoint->y = Spoint->y;
                  break; }
           setNext(s->next,&d->next,Destination); }
}
%}
```

Figure 7.9: An example of a translation definition using C

```
repA2repB data
    = ...
```

This function has a name comprising of three symbols: `repA`, the name of the representation definition to which the data values conform before translation; the symbol '2'; and `repB`, the name of the representation definition to which the data values conform after translation. One argument, in the above case called `data`, is passed to this function and corresponds to the data values to be transformed by this function. The function `repA2repB` returns the transformed data values, which are produced according to the body of the function indicated

above by the symbols '= ...'.

The Miranda translation definition given in Figure 7.10 specifies the transformation of data values conforming to the `source` representation definition, given in Figure 7.1, into data values conforming to the `destination` representation definition also given in Figure 7.1.

```
%source2destination %mira{
source2destination data
  = map src2destObj data
    where
      src2destObj ((id,0),pts)
        = Destination_choice_1 ((id, # pts, mbr), pts)
          where
            thePts = foldl dopts ((10000, 10000, -10000, -10000), []) pts
            mbr = extract_mbr thePts where extract_mbr (a,b) = a
      src2destObj ((id,1),(x,y):pts)
        = Destination_choice_2 (id, x, y)
      dopts ((mnx, mny, mxe, mxn), xys) (x,y)
          = ((min2 x mnx, min2 y mny, max2 x mxe, max2 y mxn), (x,y):xys)
%}
```

Figure 7.10: An example of a translation definition using Miranda

Section 10.3 contains a description of the author's experiences in using the C and Miranda translation environments for specifying translator modules. These experiences are also discussed in light of the author's earlier experiences using a relational translation environment.

In this Chapter a notation has been described for defining transfer specifications. A complete grammar defining the notation for specifying a transfer is given in Appendix D, and is used to produce the parser for the software tool `a2b`. Techniques are presented in Chapters 8 to 11 for generating interfaces from transfer specifications.

Construction of main interface routines

Chapter 8

An interface contains a main routine that will call, in some sequence, interface modules of the types defined in Section 6.1. Each module performs one of the transformations required to achieve the transfer. Generating the main routine from an interface definition is described in this Chapter, and generating the interface modules that will be called from the main routine is to be discussed in Chapters 9, 10, and 11.

The main routine of an interface is constructed as part of the additional C code inserted into an interface template. Selection of an interface template, comprising one or more incomplete C programs, and the additional C code to be inserted into the chosen template, are both determined by processing of the interface definition given within the transfer specification. Two different interface templates are discussed in Section 8.1 and listed in Appendix F. Methods are described in Sections 8.2 and 8.3 to generate the additional C code to be inserted into these templates.

8.1 Interface templates

Present implementation of `a2b` uses two interface templates. One forms the basis for constructing an interface which does not move data through a communications network and is presented in Appendix F.1. The other forms the basis for constructing a pair of communicating interfaces and is presented in Appendix F.2. These templates and the method of executing the interfaces which are constructed using these templates are described in Sections 8.1.1 and 8.1.2.

8.1.1 Interfaces not using communication networks

An incomplete C program forms the template of an interface which does not move data through a communications network. For the purposes of demonstration, the C program contained within this template is simple. Essentially, the program processes any command line arguments, discussed below, and then executes the sequence of data transformations constituting the main interface routine.

The interface constructed from this template is executed from a Unix command line as follows:

```
%> progName -o destDataFileName srcDataFileName
```

where

%> is the Unix command line prompt;

`progName`
: is the name of the generated interface. This name corresponds to the name of the file that contains the `a2b` specification for this interface. For example, the interface called `eg1` would have been generated from the `a2b` specification given in a file called `eg1.a2b`;

`destDataFileName`
: is the name of the file that will contain the transferred data values; and

`srcDataFileName`
: is the name of the file that contains the values to be transferred.

8.1.2 Communicating interfaces

Implementation of `a2b` permits only pairs of communicating interfaces to be generated from one transfer specification. The general approach to constructing a pair of communicating interfaces is to construct an application structured as was described earlier in Section 4.5. The application comprises a responder, which corresponds to a source communicating interface, and an initiator, which corresponds to a destination communicating interface.

The software tool `a2b` generates an application using a template comprising two incomplete C programs. One forms the basis of a responder and the other forms the basis of an initiator. For demonstration purposes, the application `imisc`, distributed with ISODE (versions 7 and 8), was modified by the author to form the template used by `a2b` for generating communicating interfaces. In Chapter 11 a more detailed discussion is given of the

`imisc` application and of the modifications made by the author to create the communicating interface template.

The pair of communicating interfaces constructed from the communicating interface template may be used to transfer data in one of two ways, corresponding to the interactive and embedded forms of an initiator supported by ISODE described earlier in Section 4.5. In the interactive form, a simple command line interface is provided by the initiator to allow a user to transfer multiple data sets. Each data transfer is performed according to a command entered by the user of the initiator. In the embedded form, the initiator is expected to be one of many components within a geographical information system. This system would execute the initiator when data values are required from another geographical information system.

To simulate an initiator embedded within a geographical information system, an initiator executed from a Unix command line can be given the arguments that would otherwise be provided by the calling geographical information system. The initiator is executed as follows:

```
%> progName X.500id getFile srcDataFileName destDataFileName
```

with the arguments being:

`progName`
: the name of the generated initiator and is formed in the same way as was described above for a non-communicating interface. The name of the generated responder is `progName-ros`. For example, given a file called `eg2.a2b` that contains an `a2b` specification comprising a communication module, the generated initiator would be called `eg2`, and the generated responder would be called `eg2-ros`;

`X.500id`
: the name of an entry in an X.500 directory. This entry contains the information necessary for the initiator to establish a communication link with the responder;

`getFile`
: the name of the command provided by the communicating interface to transfer the data;

`srcDataFileName`
: is the name of the file containing the data values to be transferred. This file name must be complete in the sense that it must begin with a '/'; and

`destDataFileName`
: the name of the file that will contain the transferred values. This file name need not

be complete, and is interpreted relative to the directory from which the initiator was executed.

When an interactive initiator is executed, only one argument is given on the command line, the name of an X.500 directory entry which describes the responder with which the initiator is to establish an association. An example of an interaction between a user and an initiator is given in Figure 8.1.

```
%>egix cn=kaka
[ using egix geodata transfer, kaka, Computer Science, ...]
...
using isode 8.0 #1 (mohua) of Wed Oct  6 14:43:20 NZDT 199
cn=kaka... connected
egix> help

Commands are:
getFile SYNOPSIS
        getFile srcFilename destFilename

DESCRIPTION
        Transfer the named source datafile to the named destination datafile

help    print this information
quit    terminate the association and exit
egix> getFile /tmp/data/eg1.s data.egix
egix> quit
%>
```

Figure 8.1: An example of an interaction with a generated interface

In Figure 8.1, the initiator is started by the command:

```
egix cn=kaka
```

where `cn=kaka` is the name of the entry in the X.500 directory which describes the required responder. The user then requests on-line help on the use of this interface by issuing the command `help`. Next, the user issues the following command:

```
getFile /tmp/data/eg1.s data.egix
```

which results in data values represented within a text-file called `eg1.s` in the directory `/tmp/data` being transferred into data values represented in a text-file called `data.egix` in the directory from which the initiator was executed. The advantage of executing the initiator in an interactive mode is that many data sets may be transferred by issuing multiple `getfile` commands, each command resulting in the transfer of a different data set.

Having described the interface templates used by a2b, the remainder of this Chapter is a description of the additional C code which is generated by a2b and inserted into the templates to form complete interfaces. The additional C code comprises data structure and variable declarations and the main interface routine. Generating data structure and variable declarations will be described in Section 8.2. Generating the main interface routine will be described in Section 8.3.

8.2 Generating data structure and variable declarations

Data structure and variable declarations are inserted into a template for each representation definition given in the transfer specification. For an interface which does not move data through a communications network, an include file `progName.h` is created which comprises statements for including other generated files containing the data structure, function, and variable declarations. For a communicating interface, two include files are generated: `progName.h` for the initiator; and `progName-ros.h` for the responder.

For example, given a representation definition called `rep` within some transfer specification, an include file would be created by the software tool a2b comprising the following statements:

```
#include "REP-ops.h"
#include "REP-types.h"
static struct type_REP_Rep *repData;
```

The file `"REP-ops.h"` is generated by the software tool `rosy`, which was described earlier in Section 4.5. This file contains the definition of error codes and the declaration of functions corresponding to the operations which are used for communication between pairs of applications. Consequently, the file `"REP-ops.h"` is of importance only when constructing communicating interfaces.

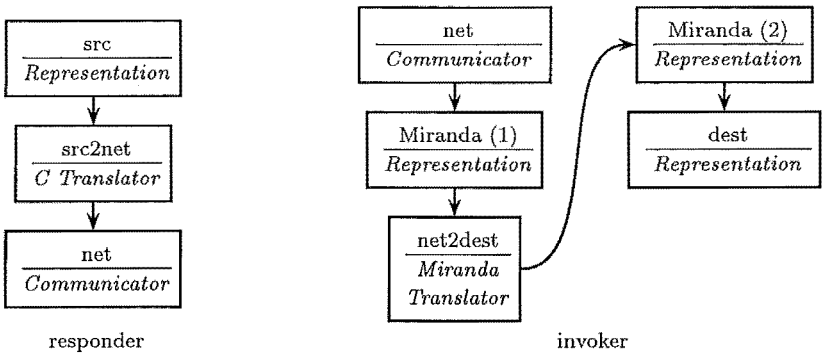
The file `"REP-types.h"` is generated by the software tool `pepsy`, described earlier in Section 4.5. This file contains the C data structure declarations which define a memory-resident representation for data values conforming to the the representation `REP`. Data values that conform to this memory-resident representation are accessible using the variable `repData`.

8.3 Generating the main interface routine

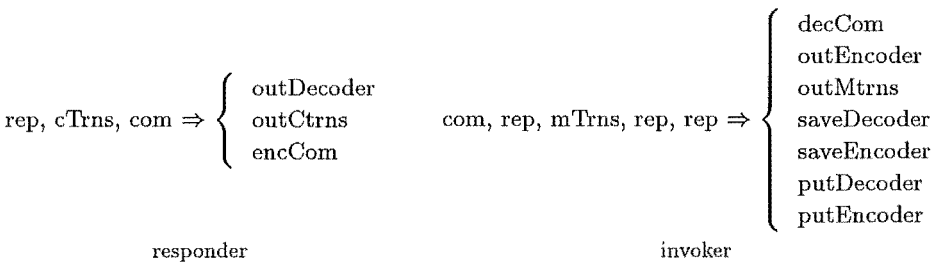
Generating the main interface routine is accomplished in three steps, as shown in Figure 8.2. They are:

```
%a2b src via net to dest
%src ...
%src2net %c{ ... %}
%net ...
%net2dest %mira{ ... %}
%dest ...
```

The transfer specification.



Step 1: Creating descriptor sequences



Step 2: Producing actions to be performed by main routines

```
srcDecode(file, &srcData);
src2net(srcData, &netData);
sendData(netData);
```

```
fd = fopen(file1, "w");
netMiraEncode(fd, netData);
fclose(fd);
shell("mira net2dest file1 file2");
destMiraDecode(file2, &destData);
fd = fopen(file3, "w");
destEncode(fd, destData);
fclose(fd);
```

responder invoker

Step 3: Implementation of the actions as C code constituting the main routines of the responder and invoker

Figure 8.2: A complete example of generating the main routines of an interface from an a2b specification

Step 1

Convert the interface definition into a sequence of module descriptors, each comprising the name and type of a module forming part of the interface. In Figure 8.2 (and later in Figure 8.3), a descriptor is shown by a box divided into two parts: the top part contains the name of the module descriptor; and the bottom part contains the type of the module descriptor;

Step 2

Convert the sequence of module descriptors into a sequence of actions to be performed as Step 3;

Step 3

Perform the actions produced by Step 2. That is, create the main interface routine comprising calls to the C functions which implement the interface modules.

8.3.1 Step 1

Step 1 is accomplished by processing the interface definition according to four rules:

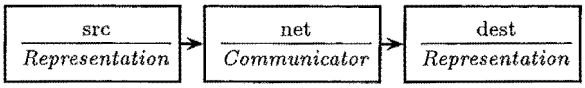
1. Insert a representation module descriptor into the sequence for each module explicitly mentioned in the interface definition;

2. Insert a translator module descriptor between any two module descriptors whose names within the interface definition are connected together by either of the keywords '`to`' or '`via`';

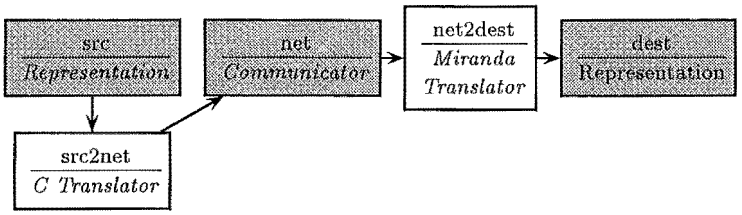
3. Insert a representation module descriptor before and after any descriptor of a translation module constructed within the Miranda translation environment. The inserted representation module descriptor refers to a representation definition that specifies an appropriate data representation for the Miranda translation environment; and

4. Start a new descriptor sequence at each communication module descriptor that occurs within the sequence.

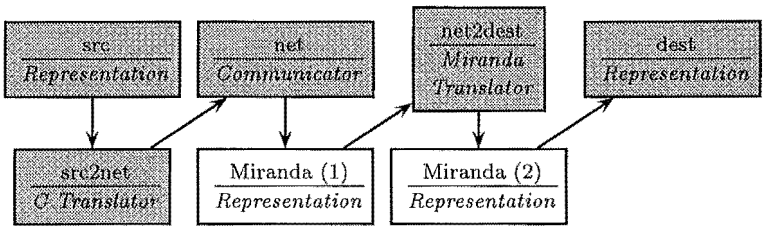
Since implementation of `a2b` permits only one pair of communicating interfaces to be constructed from a transfer specification, a descriptor sequence will, using Rule 4, be divided into at most two smaller descriptor sequences. Each of these will be used in Steps 2 and 3 described below to generate the main routine for one of the communicating interfaces. Figure 8.3 shows the application of these four Rules to the transfer specification given in Figure 8.2.



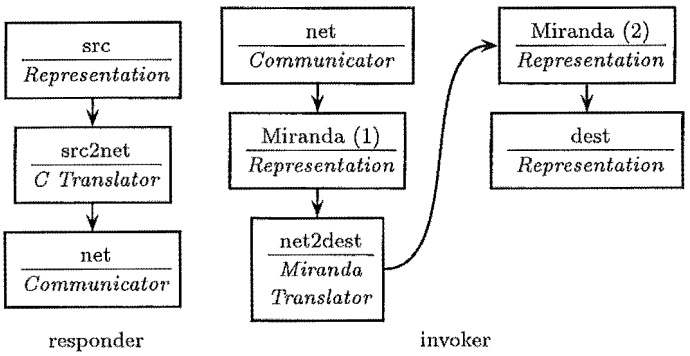
Rule 1



Rule 2



Rule 3



Rule 4

Figure 8.3: *Step 1:* Creating a descriptor sequence for the transfer specification

8.3.2 Step 2

Step 2 is the process of converting the descriptor sequence into a sequence of actions that are to be performed by a2b in Step 3 of generating the main routine of an interface, to be described in the next Section. Converting the description sequence produced in Step 1, into a sequence of actions to be performed in Step 3 is accomplished using a Mealy Machine, a finite automaton with output which is associated with state transitions. In contrast, a Moore Machine is a finite automaton with output which is associated with states.

Hopcroft & Ullman (1979) formally denote a *Mealy machine* M by a six-tuple

$$M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$$

where Q is a finite set of *states*; $q_0 \in Q$ is the *initial* state; Σ is a finite *input alphabet*; Δ is a finite *output alphabet*; δ is the *transition function* mapping $Q \times \Sigma$ to Q. That is, $\sigma(q, a)$ is a state for each state q and input symbol a; and λ is the function mapping $Q \times \Sigma$ to Δ. That is, $\lambda(q, a)$ gives the output associated with the transition from state q on input a.

The Mealy machine used during Step 2 is defined to be $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$ where $Q = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$; the initial state $q_0 = 1$, state 0 is the reject state, and state 5 is the accept state; Σ and Δ are defined in Tables 8.1 and 8.2 respectively; δ and λ are defined in Table 8.3. The transition diagram for this Mealy machine is shown in Figure 8.4 (for clarity, reject transitions are excluded from this diagram).

Input symbol $\in \Sigma$	Description
`repd`	Representation module descriptor
`cTrns`	C Translation module descriptor
`mTrns`	Miranda Translation module descriptor
`com`	Communication module descriptor
$\dashv$	End of input

Table 8.1: The input alphabet Σ

The Mealy Machine has been designed by the author in such a way that performing the actions produced as output by this machine will result in a main interface routine comprising few of the redundant encode and decode transformations discussed earlier in Section 6.2.2.1. Some redundant encode and decode transformations are included within a main interface routine because of the method used by the present version of a2b to construct Miranda translator modules. This method does not allow data values produced by one Miranda translator module to be directly processed by another Miranda translator module. Instead, they must be encoded into a text-file by the first module, and then decoded by the second.

Output symbol $\in \Delta$	Description
`outCtrns`	Insert a call to the C translator associated with this module.
`outMtrns`	Insert a call to the Miranda translator associated with this module.
`outDecoder`	Insert code to: open a data file; call the decoder associated with this module; close the data file.
`outEncoder`	Insert code to: open a data file; call the encoder associated with this module; close the data file.
`saveEncoder`	Remember the encoder associated with this module for future processing, overwriting any previously saved encoder.
`saveDecoder`	Remember the decoder associated with this module for future processing, overwriting any previously saved decoder.
`putEncoder`	Insert code to: open a data file; call the encoder that was saved for future processing by the previous output symbol saveEncoder; close the data file.
`putDecoder`	Insert code to: open a data file; call the decoder that was saved for future processing by the previous output symbol saveDecoder; close the data file.
`decCom`	Insert code to: receive encoded data from the responder; decode data from the transfer syntax.
`encCom`	Insert code to: encode data into the transfer syntax; transmit encoded data to the initiator.
`compl`	A composite action comprising the actions associated with the output symbols putDecoder and outCtrns.
ϵ	Do nothing.
`accept`	Do nothing: the routine has been completed.

Table 8.2: The output alphabet Δ

$\delta(q,a)/\ \lambda(q,a)$	Input symbols (a)				
States (q)	$\dashv$	repd	mTrns	cTrns	com
0					
1		2/outDecoder			10/decCom
2		6/outEncoder		3/outCtrns	
3		4/saveEncoder			5/encCom
4	5/putEncoder	6/putEncoder		3/outCtrns	
5	5/accept	5/accept	5/accept	5/accept	5/accept
6			7/outMtrns		
7		8/saveDecoder			
8		9/saveEncoder			11/putDecoder
9	4/putDecoder	6/ϵ		3/compl	
10		6/outEncoder		3/outCtrns	
11	5/encCom				

Table 8.3: Transition functions $\delta(q,a)$ and $\lambda(q,a)$ (for clarity, reject transitions "0/reject" are omitted)

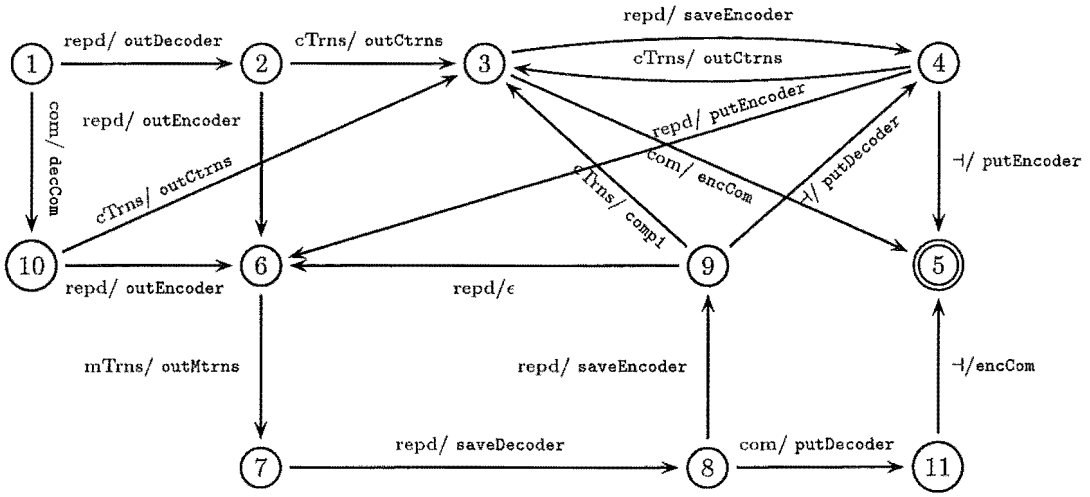


Figure 8.4: The state transition diagram for the Mealy machine used in Step 2

The method used to implement Miranda translators will be described further in Section 10.2, and examples of these redundant transformations are given below.

To illustrate Step 2, consider the transfer specification given in Figure 8.5 which specifies the transfer $\mathcal{S}_{src} \xrightarrow{*} \mathcal{I}_{int} \xrightarrow{*} \mathcal{D}_{dest}$.

```
%a2b src to int to dest
%src
...
%int
...
%dest
 ...
%src2int %c{
... %}
%int2dest %c{
... %}
```

Figure 8.5: An example transfer specification

The transfer $\mathcal{S}_{src} \xrightarrow{*} \mathcal{I}_{int} \xrightarrow{*} \mathcal{D}_{dest}$ comprises the following sequence of data transformations:

$$\mathcal{S}_{src} \xrightarrow{decode} \mathcal{T}_0 \xrightarrow{translate} \mathcal{T}_1 \xrightarrow{encode} \mathcal{I}_{int} \xrightarrow{decode} \mathcal{T}_2 \xrightarrow{translate} \mathcal{T}_3 \xrightarrow{encode} \mathcal{D}_{dest}$$

where the two translations $\mathcal{T}_0 \xrightarrow{translate} \mathcal{T}_1$ and $\mathcal{T}_2 \xrightarrow{translate} \mathcal{T}_3$ are performed by translator modules defined within the C translation environment.

As was discussed in Section 6.2.2.1, the two data sets $\mathcal{T}_1$ and $\mathcal{T}_2$ are identical. Therefore, the data transformations $\mathcal{T}_1 \xrightarrow{*} \mathcal{T}_2$ are redundant and may be removed to give a transfer

comprising the following sequence of transformations:

$$\mathcal{S}_{src} \xrightarrow{decode} \mathcal{T}_0 \xrightarrow{translate} \mathcal{T}_1 \xrightarrow{translate} \mathcal{T}_3 \xrightarrow{encode} \mathcal{D}_{dest}.$$

The descriptor sequence produced during Step 1 of processing the interface definition given above is shown in Table 8.4. Also shown in this Table are: state transitions; the actions produced as output; and the data transformations to be performed by the generated interface, when processing this descriptor sequence using the Mealy Machine.

Descriptor	Transition	Action	Transformation
`repd`	$1 \rightarrow 2$	`outDecoder`	$\mathcal{S}_{src} \xrightarrow{decode} \mathcal{T}_0$
`cTrns`	$2 \rightarrow 3$	`outCtrns`	$\mathcal{T}_0 \xrightarrow{translate} \mathcal{T}_1$
`repd`	$3 \rightarrow 4$	`saveEncoder`	$\mathcal{T}_1 \xrightarrow{encode} \mathcal{I}_{int}$
`cTrns`	$4 \rightarrow 3$	`outCtrns`	$\mathcal{T}_1 \xrightarrow{translate} \mathcal{T}_3$
`repd`	$3 \rightarrow 4$	`saveEncoder`	$\mathcal{T}_3 \xrightarrow{encode} \mathcal{D}_{dest}$
$\dashv$	$4 \rightarrow 5$	`putEncoder`	$\mathcal{T}_3 \xrightarrow{encode} \mathcal{D}_{dest}$

Table 8.4: An example of using the Mealy Machine to produce actions

Redundant encode and decode transformations are dealt with in one of two ways. In some cases the Mealy Machine does not create an action for generating source code corresponding to a redundant encode or decode transformation. In Table 8.4, for example, there is no entry for the redundant transformation $\mathcal{I}_{int} \xrightarrow{decode} \mathcal{T}_2$. In other cases, creating an action for a transformation is delayed when there is a possibility that the transformation may be redundant.

Delaying the output of an encode transformation is accomplished by using the two actions `saveEncoder` and `putEncoder`. Delaying the output of a decode transformation is accomplished using the two actions `saveDecoder` and `putDecoder`. In Table 8.4, for example, the output of an action corresponding to the data transformation $\mathcal{T}_1 \xrightarrow{encode} \mathcal{I}_{int}$ is saved, and then later forgotten because this transformation is found to be redundant. The transformation $\mathcal{T}_3 \xrightarrow{encode} \mathcal{D}_{dest}$ is also saved. However, this transformation is later found to be significant. Therefore, the action `putEncoder` is produced by the Mealy Machine to instruct `a2b` to include this transformation within the main interface routine.

As was mentioned above, when producing a main interface routine to perform a transfer containing translations specified within the Miranda translation environment, not all redundant encode and decode transformations are removed. Consider if the two translation definitions `src2int` and `int2dest` in Figure 8.5 had been defined in the Miranda translation

environment. Without any optimisation, the transfer $\mathcal{S}_{src} \xrightarrow{*} \mathcal{I}_{int} \xrightarrow{*} \mathcal{D}_{dest}$ would comprise the following sequence of transformations:

$$\mathcal{S}_{src} \xrightarrow{decode} \mathcal{T}_C \xrightarrow{encode} \mathcal{T}_{mf} \xrightarrow{decode} \mathcal{T}_{mm} \xrightarrow{translate} \mathcal{T}_{mm'} \xrightarrow{encode} \mathcal{T}_{mf'} \xrightarrow{decode} \mathcal{T}_{C'} \xrightarrow{encode} \mathcal{I}_{int}$$
$$\xrightarrow{decode} \mathcal{T}_{C''} \xrightarrow{encode} \mathcal{T}_{mf''} \xrightarrow{decode} \mathcal{T}_{mm''} \xrightarrow{translate} \mathcal{T}_{mm'''} \xrightarrow{encode} \mathcal{T}_{mf'''} \xrightarrow{decode} \mathcal{T}_{C'''} \xrightarrow{encode} \mathcal{D}_{dest}$$

An explanation of this type of transfer was given earlier in Section 6.2.2.

The optimisation of this transfer, which is performed during Step 2, is shown in Table 8.5. With this optimisation, the main interface routine, constructed according to the actions

Descriptor	Transition	Action	Transformation
repd	$1 \rightarrow 2$	outDecoder	$\mathcal{S}_{src} \xrightarrow{decode} \mathcal{T}_C$
repd	$2 \rightarrow 6$	outEncoder	$\mathcal{T}_C \xrightarrow{encode} \mathcal{T}_{mf}$
mTrns	$6 \rightarrow 7$	outMtrns	$\mathcal{T}_{mf} \xrightarrow{decode} \mathcal{T}_{mm} \xrightarrow{translate} \mathcal{T}_{mm'} \xrightarrow{encode} \mathcal{T}_{mf'}$
repd	$7 \rightarrow 8$	saveDecoder	$\mathcal{T}_{mf'} \xrightarrow{decode} \mathcal{T}_{C'}$
repd	$8 \rightarrow 9$	saveEncoder	$\mathcal{T}_{C'} \xrightarrow{encode} \mathcal{I}_{int}$
repd	$9 \rightarrow 6$		
mTrns	$6 \rightarrow 7$	outMtrns	$\mathcal{T}_{mf''} \xrightarrow{decode} \mathcal{T}_{mm''} \xrightarrow{translate} \mathcal{T}_{mm'''} \xrightarrow{encode} \mathcal{T}_{mf'''}$
repd	$7 \rightarrow 8$	saveDecoder	$\mathcal{T}_{mf'''} \xrightarrow{decode} \mathcal{T}_{C'''}$
repd	$8 \rightarrow 9$	saveEncoder	$\mathcal{T}_{C'''} \xrightarrow{encode} \mathcal{D}_{dest}$
⊣	$9 \rightarrow 4$	putDecoder	$\mathcal{T}_{mf'''} \xrightarrow{decode} \mathcal{T}_{C'''}$
⊣	$4 \rightarrow 5$	putEncoder	$\mathcal{T}_{C'''} \xrightarrow{encode} \mathcal{D}_{dest}$

Table 8.5: Another example of using the Mealy Machine to produce actions

produced by the Mealy Machine, performs the following transfer:

$$\mathcal{S}_{src} \xrightarrow{decode} \mathcal{T}_C \xrightarrow{encode} \mathcal{T}_{mf} \xrightarrow{decode} \mathcal{T}_{mm} \xrightarrow{translate} \mathcal{T}_{mm'} \xrightarrow{encode} \mathcal{T}_{mf'}$$
$$\xrightarrow{decode} \mathcal{T}_{mm''} \xrightarrow{translate} \mathcal{T}_{mm'''} \xrightarrow{encode} \mathcal{T}_{mf'''} \xrightarrow{decode} \mathcal{T}_{C'''} \xrightarrow{encode} \mathcal{D}_{dest}$$

which includes the redundant transformations $\mathcal{T}_{mm'} \xrightarrow{*} \mathcal{T}_{mm''}$ because of the method, discussed in Section 10.2, used to implement Miranda translator modules.

8.3.3 Step 3

Step 3 is the implementation of the actions associated with symbols of the output alphabet Δ within the interface template. For some symbols, implementation of the associated actions is a matter of inserting predetermined C code into the template. The output symbols for which this is the case and the C code to be inserted are given in Table 8.6.

An application layer decoder module is called from the main routine of an interface as follows:

Output symbol $\in \Delta$	C code to be inserted
outDecoder, putDecoder	`mnDecode(fileName, &mnData);`

outEncoder, putEncoder	`fd = fopen(fileName, "w");`
	`mnEncode(fd, mnData);`
	`fclose(fd);`

outCtrns	`translator(mn1Data, &mn2Data);`
outMtrns	`shell("mira translator mn1DataFile mn2DataFile");`
encCom	`sendData(mnData);`

Table 8.6: Implementation of the actions for the symbols in the output alphabet Δ

```
mnDecode(filename, &mnData);
```

where

mn

is to be replaced in the above call by the name of the representation definition;

mnDecode

is the name of the decoder;

filename

is the name of the text file in which are represented the data values to be decoded; and

mnData

is the variable in which the decoded data values will be stored.

Given a representation definition called `%source`, the decoder `sourceDecode` generated for this representation would be executed from the main interface routine by the following call:

```
sourceDecode(filename, &sourceData);
```

Presentation layer decoders are called from within communicator modules.

An application layer encoder module is called from the main routine of an interface as follows:

```
mnEncode(fd, mnData);
```

where

mn

is to be replaced in the above call by the name of the representation definition;

`mnEncode`

is the name of the decoder;

`fd`

is the file descriptor which points to the text file into which the data values are to be encoded; and

`mnData`

is the variable in which are represented the data values to be encoded.

Before the encoder is called, the file descriptor `fd` must be initialised as follows:

```
fd = fopen(filename, "w");
```

where `filename` is the name of either the destination text file or a temporary text file created by the interface. After calling the encoder module the file must be closed as follows:

```
fd = fclose(fd);
```

To illustrate the use of an encoder module, data conforming to a representation called `source` would be encoded from the main routine of an interface as follows:

```
fd = fopen(filename, "w");
sourceEncode(fd, sourceData);
fd = fclose(fd);
```

Presentation layer encoders are called from within communication modules.

The C code shown in Figure 8.6 for implementing the actions associated with the output symbol `encCom` comprises a call to a routine called `sendData` which performs the following data transformations:

$$\begin{array}{ll} \text{Presentation layer} & \mathcal{T}_S \overset{encode}{\longrightarrow} \mathcal{N}_S \\ \text{Layers 1–5} & \mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D \end{array}$$

The routine `sendData`, discussed further in Section 11.2.3.2, comprises calls to a presentation layer encoder module and a communications module. The C code for implementing the actions associated with the output symbol `decCom` is explained in Section 11.2.3.4, and correspond to calling a communicator module and a presentation layer decoder module to perform the following transformations:

$$\begin{array}{ll} \text{Presentation layer} & \mathcal{N}_D \overset{decode}{\longrightarrow} \mathcal{T}_S \\ \text{Layers 1–5} & \mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D \end{array}$$

Implementation of the actions associated with the output symbol `comp1` is simply the combined implementation of the actions associated with the output symbols `putDecoder` and `outCtrns`. Therefore, `comp1` means implement the most recently saved decoder transformation and implement the C translation corresponding to this action. The output symbols `saveDecoder` and `saveEncoder` are directives to `a2b` and, as was described in the preceding Section, result in no source code being generated for the main interface routine.

The next Chapter contains descriptions of generating and constructing the decoder and encoder modules that are called from the main routine of an interface.

Construction of decoder and encoder modules

Chapter 9

Decoder and encoder interface modules transform the physical representation of data values. In Section 6.1 the distinction was made between presentation layer encoder and decoder modules and application layer encoder and decoder modules. Construction of presentation layer encoder and decoder modules is discussed together with the construction of communicator modules in Chapter 11 because all three modules are constructed using the ISO Development Environment (ISODE).

Application layer decoder and encoder modules are constructed according to the representation definitions of a transfer specification, as shown in Figure 9.1.

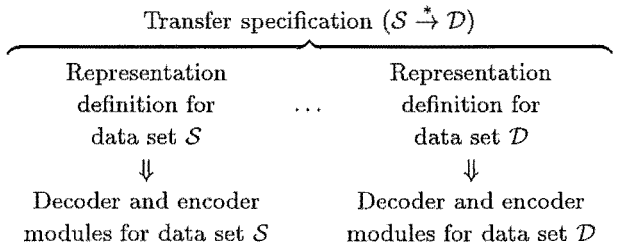


Figure 9.1: Generating encoder and decoder modules from a transfer specification

Generating decoder modules is discussed in the next Section and generating encode modules will be discussed in Section 9.2.

9.1 Decoders

Pascoe & Penny (1990) suggested an analogy between the parsing by a compiler of a program written in some computer language, and the parsing by a decoder of geographical data values represented within a text file. The author exploited this similarity by using the compiler-building tools `yacc` (Johnson 1979) and `lex` (Lesk & Schmidt 1979) to construct a parser and a scanner which performs the function of a decoder. That is, parse the text file representation of geographical data values and represent them within a relational database.

The method adopted in this thesis is different from the author's earlier approach for two reasons. First, the decoded data values are represented using C data structures rather than relations in a relational database. This was in part to allow the decoders to be combined with communication modules which send and receive data through networks using ISODE, and in part to allow experimentation with other translation environments. Second, and less significantly, the software tools `bison` (Corbett 1989) and `flex` (Paxson 1990) are used rather than `yacc` and `lex`. Use of `bison` and `flex`, was preferred by the author because of the better debugging facilities provided within the software generated by `bison` and `flex`.

In the approach to constructing decoders defined here, a decoder module comprises three components:

a scanner
: divides a sequence of ASCII characters into smaller sequences called tokens;

a parser
: recognises various sequences of tokens as being particular types of data values and creates memory-resident representations for these values; and

C data structure declarations
: define the memory-resident representation of data decoded by the parser.

As shown in Figure 9.2, all three components are generated from definitions which are themselves generated from a representation definition.

The software tool `a2b` is used to generate definitions of the required scanner, parser, and abstract syntax from a representation definition. Definitions of the scanner, parser, and abstract syntax are processed by `flex`, `bison`, and `pepsy` (Rose *et al* 1991) to generate, respectively, the scanner, parser and C data structure declarations. Construction of the scanner, parser and C data structure declarations is now discussed.

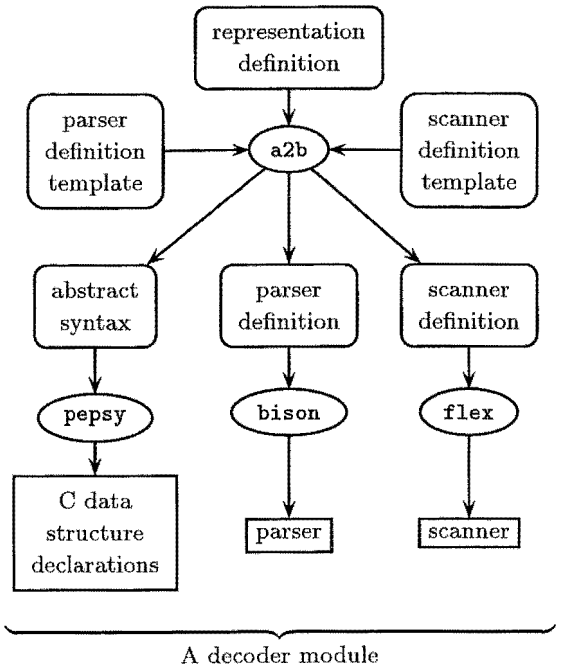


Figure 9.2: Summary of the construction of a decoder

9.1.1 Scanners

A scanner divides a text file into tokens and is generated by `flex` from a scanner definition comprising tokens defined by regular expressions. A scanner definition is generated by `a2b` for each representation definition defined within a transfer specification, as shown in Figure 9.3.

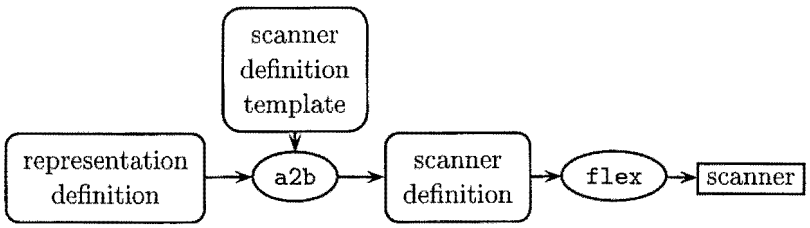


Figure 9.3: Generating a scanner from a representation definition

The software tool `a2b` combines the default text file representation of each simple data type with any literals and user-defined text file representations (see Section 7.2.2.1), and inserts the corresponding token definitions into a scanner definition template which is presented in Appendix F.3.1. A token definition comprises a regular expression and the actions

to be performed by the scanner on recognition of the token. Token definitions for the simple data types `string`, `real`, and `integer` are shown in Table 9.1, together with the standard definition for any literal.

	Token definition	
Data type	Regular expression	Scanner actions
integer	`[+-]?[0-9]+`	`(*yylvalp).integer = atol(yytext);` `return CASINTEGER3;`

string	`\"((\\.)|[^\"\\])*\"`	`(*yylvalp).string = ( yytext ?`
		`str2qb(yytext+1,strlen(yytext)-2,0) :`
		`NULL);`
		`return CASSTRING30;`

real	`[+-]?[0-9]*\.[0-9]+\` `([Ee][-+][0-9][0-9])?`	`(*yylvalp).real = atof(yytext);` `return CASREAL31;`
literal	`"literal value"`	`return CASTOKEN19;`

Table 9.1: Token definitions for simple data types, and literals

An action performed by the scanner for the simple data types `string`, `real`, and `integer` is the conversion of the sequence of characters forming the token, into an appropriate memory-resident representation. These conversions are performed by the functions:

`atol()`
: a standard C library function for converting a token into a long integer value;

`str2qb()`
: an ISODE library function for converting a token into an ISODE memory-resident representation for a string value; and

`atof()`
: a standard C library function for converting a token into a real value.

The value produced by any of these functions is stored in the variable called `yylvalp`, which is defined by the parser. The last action performed by a scanner for any token is to return an integer value that uniquely identifies the token read. In Table 9.1, these integer values have the symbolic names `CASINTEGER3`, `CASSTRING30`, `CASREAL31`, and `CASTOKEN19`. However, these symbolic names vary for each scanner generated by `a2b`.

9.1.1.1 Special techniques

In Section 7.2.2.1, a notation was defined for specifying a new data type comprising values which are represented in a text file using a fixed number of characters. For example, an

integer data type comprising values which are represented using only 3 characters is defined by the following notation:

```
threeCharInt :  integer:3;
```

Generating a scanner to identify both fixed and variable length tokens is accomplished by defining a scanner that may operate in one of three modes: initial, free, and fixed. Any token definition inserted into the scanner definition by `a2b` is prefixed by a label `<sc>`, indicating that the token definition is only active when the scanner is in the start condition called `sc`. More information regarding `flex` and start conditions is available in the UNIX on-line man pages.

Scanner definitions of variable-length tokens are prefixed by the label `<free>`. For example, the entry in a scanner for a variable-length token representing an integer would be:

```
<free>[+-]?[0-9]+ { (*yylvalp).integer = atol(yytext);
                    return CASINTEGER3; }
```

Scanner definitions of fixed-length tokens are dealt with by one entry in the scanner definition, which is prefixed by the label `<fixed>`. This entry reads the expected number of characters and then transforms the characters into a memory-resident value appropriate for the expected type of the token. The expected size and type of a fixed-length token, and setting the scanner in the correct mode of operation are responsibilities of the parser, as will be explained in Section 9.1.2.1.

9.1.2 Parsers

A parser divides the sequence of tokens produced by a scanner into different types of structured data values, and represents these values in memory using C data structures. The collection of tokens forming the different types of structured data values is defined by a context-free grammar. This grammar is inserted by `a2b` into a parser definition template, presented in Appendix F.3.2, to form a parser definition which is processed by `bison` to generate a parser, as shown in Figure 9.4.

The context-free grammar forming the parser definition comprises a collection of productions, similar to BNF productions, that define the sequence of tokens that correspond to different types of structured data values. A production expressed using the notation processed by `bison` is similar to a BNF production, but differs in the following ways:

- the left- and right-hand sides of a bison production are separated by the symbol ':' instead of the symbol $\longrightarrow$ used within a BNF production;

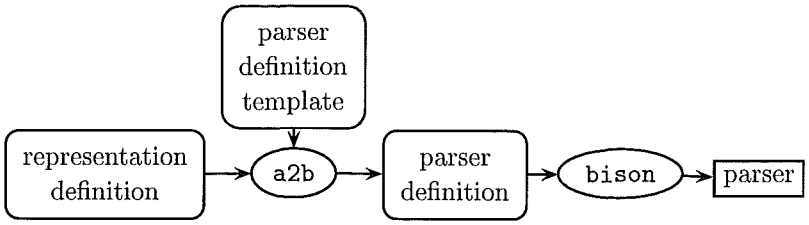


Figure 9.4: Generating a parser from a representation definition

- a bison production is terminated by the symbol ';';
- a token name is used within a bison production instead of the literals given in a BNF production; and
- names occurring on the right-hand side of a bison production are not enclosed within the symbols ⟨ and ⟩.

To illustrate the differences between bison and BNF productions, a sequence of integers preceded by the literal "`Seq of integers:`" would be defined by the following bison productions:

```
seqOfInt : STRINGLITERAL intSeq;
intSeq   : INTEGER | intSeq INTEGER;
seqOfInt : STRINGLITERAL intSeq;
```

where `STRINGLITERAL` and `INTEGER` are tokens defined in the scanner definition as follows:

```
"Seq of integers:"      { return STRINGLITERAL; }
[0-9]+                  { .... return INTEGER; }
```

The symbols ... indicate where part of the action has been omitted. The BNF productions equivalent to the bison productions given above are:

⟨seqOfInt⟩ ⟶ `Seq␣of␣integers:` ⟨intSeq⟩
⟨intSeq⟩ ⟶ ⟨integer⟩ | ⟨intSeq⟩ ⟨integer⟩
⟨integer⟩ ⟶ ⟨digit⟩ | ⟨integer⟩ ⟨digit⟩
⟨digit⟩ ⟶ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

The software tool `a2b` generates bison productions for recognising the different types of structured data values specified within a representation definition. The different types of bison productions corresponding to the structured type constructs described earlier in Section 7.2.1 are shown in Table 9.2.

Structured data type	Bison productions
`setValues : {val1 val2 val3};`	`setValues : val1 val2 val3;`
`aSetOf : {val}*;`	`aSetOf : val | aSetOf val;`
`anotherSetOf : {val || "sep"}*;`	`anotherSetOf : val | anotherSetOf sepTkn val;`
`seqValues : (val1 val2 val3);`	`seqValues : val1 val2 val3;`
`aSetOf : (val)*;`	`aSeqOf : val | aSeqOf val;`
`aChoice : < val1 | val2 >;`	`aChoice : val1 | val2;`
`enum : << "str1" | "str2" >>;`	`enum : str1tkn | str2tkn;`

Table 9.2: Examples of the a2b constructs for specifying structured data types, and the corresponding bison productions that are inserted into the parser definition template

In Section 9.1.2.1, bison productions are described for recognising structured data values of the type defined using the N-Sequence or N-Set constructs defined earlier in Sections 7.2.1.2 and 7.2.1.3. Also in Section 9.1.2.1 is an explanation of the special technique used by the generated parser for dealing with integer, real, or string data values that are represented by a fixed number of characters.

Once a token sequence defined by a bison production is recognised, one or more actions are performed by a parser. These actions are enclosed with braces and are expressed using the C programming language. The actions generated by `a2b` instruct the parser to construct a memory-resident representation of the data values that were represented in the text file by the recognised tokens. The memory-resident representation comprises C data structures according to the declarations that were generated by `pepsy`. Generating these declarations is to be explained in Section 9.1.3.

To illustrate the general form of a bison production and the actions associated with a production, consider a representation definition that includes the following type definition:

```
point: (x y);
```

The software tool `a2b` would generate the following bison production:

```
point : x y
      {
          $$ = malloc(sizeof(*$$));
          $$->x = $1;
          $$->y = $2;
      }
      ;
```

where `bison` replaces `$$`, `$1`, and `$2` with the names of variables. The variable corresponding to `$$` will point to a C data structure in which are represented the x and y coordinate values

of the point. The variable corresponding to `$1` will point to the value associated with the first name on the right-hand side of the production, which in this case is the x coordinate value produced by the scanner. Similarly, `$2` will point to the y coordinate value.

9.1.2.1 Special techniques

The generated parsers incorporate special techniques for dealing with fixed-sized tokens, which are data values represented by a fixed number of characters within a text file, and for dealing with N-Sets and N-Sequences, which are collections of n data values all of which are of the same type. These techniques are now described.

Parsers are constructed by `a2b` to set the scanner in the correct mode for reading the next token before requesting this token from the scanner. A scanner may be set into one of two modes (Section 9.1.1.1): free, if the size and type of the next token is unknown; or fixed, if the size and type of the next token is known. In setting the scanner mode to be fixed, the parser must also indicate the number of characters that comprise the next token, and the type of value represented by these characters.

A parser which sets the mode of the scanner is defined by inserting a special non-terminal in front of tokens that form bison productions within the parser definition. For example, in the following production:

```
aProduction : setModeFree INTEGER;
```

the token `INTEGER` is preceded by the non-terminal `setModeFree` which is defined by the following production:

```
setModeFree : SETFLEXMODE { scannerMode = free; };
```

where

`SETFLEXMODE`
: is the token returned by the scanner in response to a parser's request for a token when the mode of the scanner is neither free nor fixed. In returning this token, the scanner reads no characters from the text file;

`scannerMode = free;`
: is the action which sets the scanner into the correct mode for reading the next token. In this example, the correct mode is free, and the next token, `INTEGER`, is an integer value represented by an unknown number of characters. The parser will perform this action after the scanner returns the token `SETFLEXMODE`, and before requesting the next token, which is expected to be an `INTEGER`.

When parsing the text file according to the productions `aProduction` and `setModeFree`, the initial mode of the scanner is neither free nor fixed. Therefore, the scanner returns the token `SETFLEXMODE` in response to the parser's request for a token. This token corresponds to the right-hand side of the production `setModeFree`. Consequently, the parser performs the action associated with this production, and sets the scanner into the free mode. The parser then requests the next token, and the scanner matches a sequence of characters to the regular expression defining the token `INTEGER`, modifies the scanning mode from being free to being neither fixed nor free, and returns this token. All symbols on the right-hand side of the production `aProduction` have been processed and parsing is complete.

Parsing a fixed-length value is accomplished in a similar way. For example, parsing of an integer value represented by 3 characters is defined by the following productions:

```
bProduction      : setModeInteger3 INTEGER;
setModeInteger3 : SETFLEXMODE
                  {
                   scannerMode = fixed;
                   tokenSize = 3;
                   tokenType = integer;
                  };
```

and is similar to that described above for the productions `aProduction` and `setModeFree`. However, the actions associated with the non-terminal `setModeInteger3` set the scanner into the fixed mode and instructs the scanner to return an integer value constructed from reading the next 3 characters.

The technique described above for setting the mode of the scanner relies on the method of parsing used by the parsers generated by the software tools `bison` and `yacc`. The technique may not work if some other parser generating tool is used. Present implementation of this technique in `a2b` relies on notation within a representation definition which should not be required. This additional notation is described in Appendix D.2. Discussion now moves to the technique incorporated within the generated parsers for decoding values of either the N-Set or N-Sequence data types.

Pascoe (1989) described a technique for decoding enumerated repeating groups using the software tool `yacc`. Enumerated repeating groups discussed by Pascoe (1989) are equivalent to values of both the N-Set or N-Sequence data types. Consequently, the same technique is used within the parsers generated by `a2b`.

As explained in Sections 7.2.1.2 and 7.2.1.3, there are two values associated with an N-Sequence or N-Set value: the integer value n, and a collection of n values. Although the value n must occur before the collection of values, many other unrelated values may be

represented between n and the associated collection of values. The technique used by the generated parser and scanner for decoding N-Set and N-Sequence values is as follows.

Once decoded, the value n is is stored in a counter to allow for any values between the value n and its associated collection of values. Just before requesting the first of the collection of values, the parser initialises a counter used by the scanner with the stored value of n. Each time a token is requested by the parser, the scanner checks to see if a collection of values associated with an N-Set or N-Sequence is being decoded and, if so, whether the last value of this collection has been decoded. When the last of a collection of values has been decoded, the `EORG` (End Of Repeating Group) token is returned by the scanner in response to the parser's next request for a token. Otherwise, the scanner returns tokens in the usual manner, and the parser decrements the scanner's counter for each value of the collection decoded. When the parser receives the `EORG` token, the collection of values has been decoded, and the parser begins to decode the next value.

An example of an N-Set is:

```
3 Points:  10998, 44532, 66576
```

where: n is `3`; the collection of three values is `10998`, `44532`, and `66576`; and the literals "`Points:`" and "`,`" are part of the value's representation. This type of N-Set value is defined in Figure 9.5(a) using the notation described in Chapter 7.2.1.3.

The bison productions generated by `a2b` from this type definition are shown in Figure 9.5(b) where

`casCounter[0] = $1;`

is the action to store the value n, the preceding `INTEGER` token, in a counter;

`INITRG(casCounter[0]);`

is the action which initialises the scanner's counter to the value previously stored in the counter called `casCounter[0]`;

`EORG`

is the token that will be generated by the scanner once the collection of values has been decoded; and

`DECRG;`

is the action for decrementing the scanner's counter, and is performed each time a value within the collection is decoded.

Note that in Figure 9.5, the non-terminals for setting the mode of the scanner have been omitted for clarity.

```
value  : ( #n "Points:" { val || ","}^n );
n, val : integer;
```

(a) An N-Set type definition

```
value :       INTEGER { casCounter[0] = $1; } CASTOKEN0 collection  {
              /* create memory representation for value */
      }
      ;
collection :  { INITRG(casCounter[0]); } cValues EORG  {
              /* pass value on */
      }
      ;
cValues :     INTEGER  {
              /* create memory representation */
              DECRG;
      }
      |       cValues CASTOKEN1 INTEGER  {
              /* create memory representation; *
               * add to previous values        */
              DECRG;
      }
      ;
```

(b) Bison productions generated by a2b for the type definition given in 9.5(a)

Figure 9.5: An N-Set type definition and the corresponding bison productions

9.1.3 C data declarations

A decoder transforms data values represented in a text file into values having a memory-resident representation using C data structures. This memory-resident representation created by the parser is determined by C data structure declarations which are generated from a representation definition as shown in Figure 9.6.

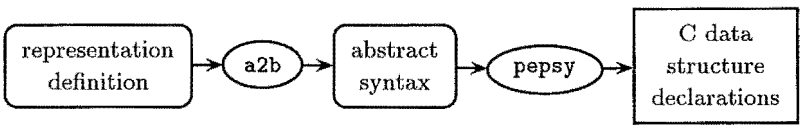


Figure 9.6: Generating C data structure declarations from a representation definition

The software tool a2b generates an abstract syntax from which the C data structure declarations are generated by pepsy. The software tool pepsy is a part of the ISO Development Environment, as was described in Section 4.5, and generates C data structure declarations

for representing the types of data values defined by an abstract syntax using ASN.1. ASN.1 will be described in Appendix A.2. In Figure 9.7(a), the abstract syntax generated by `a2b` is shown for the representation `%source` defined in Figure 7.2 on page 99. Figure 9.7(b) shows the C data structure declarations generated by `pepsy` from the abstract syntax given in Figure 9.7(a).

In general, sets and sequences comprising values of one type are represented by a linked list of C data structures, each structure representing one value of the set or sequence. A set or sequence comprising a fixed number of different types of data values is represented by a C data structure comprising a collection of variables, one for each data value of the set or sequence.

A choice between different types of values is represented by a C data structure comprising two components: a union structure, comprising variables in which values of the types defined by the choice may be represented; and a tag variable, to indicate which type of value was chosen, and hence the variable of the union structure in which the value is represented. More information describing `pepsy` can be found in the ISODE User's manual (Rose *et al* 1991) and other technical documents provided with the `pepsy` source code.

Table 9.3 shows different A2B constructs described earlier in Section 7.2 for defining various data types and the corresponding ASN.1 abstract syntax which is generated by `a2b`.

Specifying a type comprising a set of one or more values within a representation definition results in `a2b` generating an abstract syntax defining a data type as being a set comprising two components: a single value corresponding to the mandatory value, and a set of zero or more values. A sequence of one or more values is dealt with by `a2b` in a similar manner, with an ASN.1 type definition of a sequence being created which also consists of two components: a single value corresponding to the mandatory value, and a sequence of zero or more values. Examples of these sets and sequences are given in Table 9.3.

Specifying an enumerated type within a representation definition results in `a2b` generating an ASN.1 integer data type definition. Values of this type correspond to the string values given in the enumerated type definition. For example, in Table 9.3 the first value `"value A"` of the enumerated type `enumType` would be represented by the integer value of 0, the second `"value B"` would be represented by 1, and so on.

Construction of a decoder module from a representation definition was summarised earlier in Figure 9.2. The software tool `a2b`, reads a transfer specification, and for each representation defined within this specification creates a scanner definition, a parser definition, and an abstract syntax. These are processed by `flex`, `bison`, and `pepsy` to generate the scanner, the parser, and the C data structure declarations that form a decoder.

```
SOURCE DEFINITIONS ::=
BEGIN
Source  ::= [ APPLICATION 1 ] SEQUENCE OF Feature
Feature ::= [ APPLICATION 2 ] SEQUENCE {
        casS0           [ 0 ]   CasS0,
        casS2           [ 1 ]   CasS2
}
Point   ::= [ APPLICATION 3 ] SEQUENCE {
        x               [ 0 ]   INTEGER,
        y               [ 1 ]   INTEGER
}
CasS0   ::= [ APPLICATION 4 ] SEQUENCE {
        id              [ 0 ]   INTEGER,
        type            [ 1 ]   INTEGER
}
CasS2   ::= [ APPLICATION 5 ] SEQUENCE OF Point
END
```

(a) The abstract syntax generated by a2b

```
struct type_SOURCE_Source {
    struct type_SOURCE_Feature *element_SOURCE_0;
    struct type_SOURCE_Source *next;
};
struct type_SOURCE_Feature {
    struct type_SOURCE_CasS0 *casS0;
    struct type_SOURCE_CasS2 *casS2;
};
struct type_SOURCE_Point {
    integer     x;
    integer     y;
};
struct type_SOURCE_CasS0 {
    integer     id;
    integer     type;
};
struct type_SOURCE_CasS2 {
    struct type_SOURCE_Point *element_SOURCE_1;
    struct type_SOURCE_CasS2 *next;
};
```

(b) The C data structure declarations generated by pepsy from the abstract syntax given in 9.7(a)

Figure 9.7: The abstract syntax and C data structure declarations generated from the %source representation definition in 7.2

A2B type construct	ASN.1 type definition

aSet : {a b c};	ASet ::= [APPLICATION 2]
	SET {
	a [0] INTEGER,
	b [1] REAL,
	c [2] IA5String
	}

aSequence : (a b c);	ASequence ::= [APPLICATION 3]
	SEQUENCE {
	a [0] INTEGER,
	b [1] REAL,
	c [2] IA5String
	}

aSetof : {a}*;	ASetof ::= [APPLICATION 4] SET OF INTEGER
aSequenceOf : (a)*;	ASequenceOf ::= [APPLICATION 5] SEQUENCE OF INTEGER

aSet1More : {a}+;	ASet1More ::= [APPLICATION 6]
	SET {
	a [0] INTEGER,
	casSO [1] CasSO
	}
	CasSO ::= [APPLICATION 9]
	SET OF INTEGER

aSequence1More : (a)+;	ASequence1More ::= [APPLICATION 7]
	SEQUENCE {
	a [0] INTEGER,
	casS2 [1] CasS2
	}
	CasS2 ::= [APPLICATION 10]
	SEQUENCE OF INTEGER

choice : < a | b | c>;	Choice ::= [APPLICATION 8]
	CHOICE {
	a [0] INTEGER,
	b [1] REAL,
	c [2] IA5String
	}

enumType : << "value A" | "value B" | "value C" >>;	enumType ::= [APPLICATION 8] INTEGER

Table 9.3: The A2B type constructs and their ASN.1 equivalents generated by the software tool a2b

9.2 Encoders

An encoder module is divided into three components:

encoding functions
: which encode the values as a sequence of characters in a text file;

a delimiter lookup table
: which determines whether a TAB, space, or some other delimiter specified within the representation definition is to be inserted between a pair of encoded data values; and

C data structure declarations
: which define the memory-resident representation of the data values to be encoded into a text file.

The process of generating these three components of an encoder module is shown in Figure 9.8. Construction of encoding functions is described in the next Section and the con-

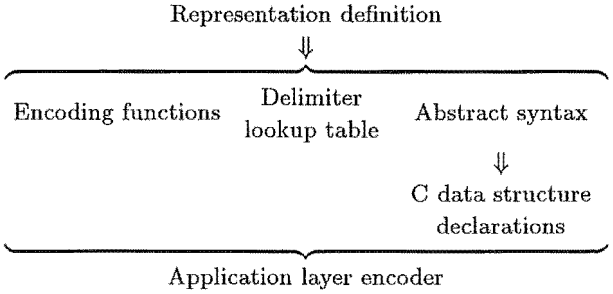


Figure 9.8: Generating application layer encoders

struction of delimiter lookup tables is described in Section 9.2.2. The data structure declarations for an encoder are constructed in the same way that C data structure declarations are constructed for decoders, which was described in Section 9.1.3.

9.2.1 Encoding functions

Encoding functions encode data values into a text file according to a representation definition. An encoding function is generated by `a2b` for each type of data value specified within the representation definition. The process of generating encoding functions is shown in Figure 9.9. Encoding functions comprise:

- calls to the standard C library function `fprintf`, to insert literals that are defined as a part of the data type; and

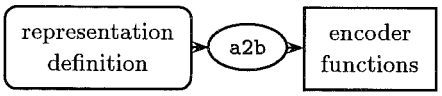


Figure 9.9: Generating encoder functions from a representation function

- calls to other functions that encode values of component data types, using either encoding functions defined for these components by the representation definition, or encoding functions provided by `a2b` for the simple data types `integer`, `real`, and `string`.

Figure 9.10 shows some of the encoding functions generated by `a2b` for the `%source` representation definition given in Figure 7.2.

Encoding functions use another function called `doKerning` to determine whether a delimiter is needed between the previously encoded data value and the value about to be encoded. Delimiter insertion was discussed in Section 7.2.2.3. The function `doKerning` decides according to the content of a delimiter lookup table which is constructed as described below.

9.2.2 Delimiter lookup tables

The function `doKerning` determines whether a delimiter is required between the text file representations of two data values according to a delimiter lookup table which was generated by `kerngen`. The software tool `kerngen` is described in Appendix C. The process of generating a delimiter lookup table from a representation definition is shown in Figure 9.11.

The software tool `kerngen` examines both the finite state automata included within the scanner generated by `flex` (Section 9.1.1), and any explicit delimiter insertion definitions given within the representation definition (Section 7.2.2.3). When `kerngen` determines that the text file representation of consecutive data values need to be separated, an entry is inserted into the delimiter lookup table. This entry comprises the two type names of the data values and the delimiter to be inserted.

Before encoding any data value, the delimiter insertion function `doKerning` is called to search the delimiter lookup table for an entry comprising the names of the data type of the value about to be encoded, and of the previously encoded data value. When an entry is found, the required delimiter is inserted by `doKerning` and control is returned to the function which encodes the data value.

In summary, the encoding functions, delimiter lookup table, and the C data structure declarations which form an encoder module are generated from a representation definition using `a2b`, `flex`, `kerngen`, and `pepsy`. The next Chapter is a description of constructing translator modules.

```
static featureEncode(fd, data) FILE *fd; struct type_SOURCE_Feature *data;
{
        doKerning(fd, CASTOKEN1); fprintf(fd, "feat");
        casS0Encode(fd, data->casS0);
        doKerning(fd, CASTOKEN2); fprintf(fd, "\ncoor");
        casS2Encode(fd, data->casS2);
}
static casS2Encode(fd, data) FILE *fd; struct type_SOURCE_CasS2 *data;
{
        for(;data;data=data->next) {
                pointEncode(fd,data->element_SOURCE_1);
        }
}
static pointEncode(fd, data) FILE *fd; struct type_SOURCE_Point *data;
{
        CASINTEGER3Encode(fd, data->x);
        CASINTEGER3Encode(fd, data->y);
}
static CASINTEGER3Encode(fd,data) FILE *fd; integer data;
{
        doKerning(fd, CASINTEGER3);     fprintf(fd, "%d", data);
}
static typeEncode(fd, data) FILE *fd; integer data;
{
        switch(data) {
                case 0:
                        doKerning(fd, CASTOKEN4); fprintf(fd, "L");
                        break;
                case 1:
                        doKerning(fd, CASTOKEN5); fprintf(fd, "P");
                        break;
                default:;
        }
}
```

Figure 9.10: Encoding functions generated by a2b for the %source representation definition in 7.2

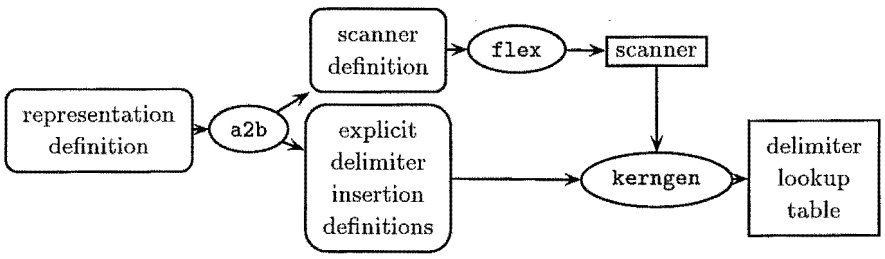


Figure 9.11: Generating a delimiter lookup table from a representation definition

Construction of translator modules

Chapter 10

A *translator module* translates data values to conform to a different conceptual schema, a different implementation schema, or both. A translator module that transforms some set of data values $\mathcal{T}_0$ into another data set $\mathcal{T}_1$, conforming to different conceptual and implementation schemas, is described as:

$$\mathcal{T}_0(C_0,\ I_0,\ P_0) \xrightarrow{*} \mathcal{T}_1(C_1,\ I_1,\ P_1).$$

Note that transforming a data set to conform to different conceptual or implementation schemas also results in the transformed values having a different physical representation.

As explained in Section 7.3.1, a translation definition comprises functions which are expressed using either the C or Miranda programming languages. In this Chapter, an explanation is given of the method by which `a2b` constructs a translator module from a translation definition. In particular, constructing C translator modules from C translation definitions is explained in Section 10.1, and constructing Miranda translator modules from Miranda translation definitions is explained in Section 10.2.

The Chapter is concluded with a discussion of the author's experiences in constructing translator modules using the translation environments provided by the C and Miranda programming languages. These experiences are compared to those gained during previous research by the author in which a relational DBMS was used as a translation environment.

10.1 Generating C translator modules

Inclusion of a C translator module within an interface results in a transfer that comprises the following sequence of transformations:

$$\mathcal{S} \xrightarrow{*} \mathcal{T}_C \overset{translate}{\longrightarrow} \mathcal{T}_{C'} \xrightarrow{*} \mathcal{D}$$

where $\mathcal{T}_C$ and $\mathcal{T}_{C'}$ are data sets represented in memory using data structures provided by the C programming language, and the transformation $\mathcal{T}_C \overset{translate}{\longrightarrow} \mathcal{T}_{C'}$ is performed by a C translator module.

A C translator module is comprised of the C functions provided within the C translation definition and statements generated by `a2b` for including the C data structure declarations to which the data sets $\mathcal{T}_C$ and $\mathcal{T}_{C'}$ conform. One of these functions given in the C translation definition is to be called from the main routine of the interface and must have the general form that was defined in Section 7.3.1.1. The C data structure declarations to which the data sets $\mathcal{T}_C$ and $\mathcal{T}_{C'}$ conform are generated by `pepsy` using the methods described earlier in Section 9.1.3. An example of a C translator module is given in Figure 10.1. This module was generated by `a2b` from the C translator definition given in Figure 7.9 on page 115.

10.2 Generating Miranda translator modules

Inclusion of a Miranda translator module within an interface results in a transfer that comprises the following sequence of transformations:

$$\mathcal{S} \xrightarrow{*} \mathcal{T}_C \overset{encode}{\longrightarrow} \mathcal{T}_{mf} \overset{decode}{\longrightarrow} \mathcal{T}_{mm} \overset{translate}{\longrightarrow} \mathcal{T}_{mm'} \overset{encode}{\longrightarrow} \mathcal{T}_{mf'} \overset{decode}{\longrightarrow} \mathcal{T}_{C'} \xrightarrow{*} \mathcal{D} \qquad (10.1)$$

where $\mathcal{T}_C$ and $\mathcal{T}_{C'}$ are data sets having different memory-resident representations using C data structures, $\mathcal{T}_{mf}$ and $\mathcal{T}_{mf'}$ are data sets having different text file representation using Miranda data structures, and $\mathcal{T}_{mm}$ and $\mathcal{T}_{mm'}$ are data sets having different memory-resident representations using Miranda data structures.

The combination of modules performing in the sequence of transformations (10.1) was described in Section 6.2.2. The transformations $\mathcal{T}_C \overset{encode}{\longrightarrow} \mathcal{T}_{mf}$ and $\mathcal{T}_{mf'} \overset{decode}{\longrightarrow} \mathcal{T}_{C'}$ are performed by an encoder module and a decoder module. These modules are generated by `a2b`, in the manner described in Chapter 9, from representation definitions which are also generated by `a2b` as will be explained in Section 10.2.1. The transformations $\mathcal{T}_{mf} \overset{decode}{\longrightarrow} \mathcal{T}_{mm} \overset{translate}{\longrightarrow} \mathcal{T}_{mm'} \overset{encode}{\longrightarrow} \mathcal{T}_{mf'}$ are performed by functions within the Miranda translator module.

```
#include <stdio.h>
#include "SOURCE-types.h"
#include "DESTINATION-types.h"
#include "define.h"
source2destination(s,dp)
     struct type_SOURCE_Source *s;
     struct type_DESTINATION_Destination **dp;
{
      Ddecl(Destination, *d); Sdecl(CasS2, *sP); Ddecl(CasS2, *dP);
      new(&d,Destination);*dp = d;
      for (;s;s=s->next,d=d->next) {
            new(&Dfeature,CasS8);
            switch (Sfeature->casS0->type) {
               case 0 : { /* Line */
                     int init = 1;
                     Dfeature->offset = type_DESTINATION_CasS8_casS4;
                     new(&Dline,CasS4); new(&DlineInfo,CasS0);
                     DlineInfo->id = Sfeature->casS0->id;
                     DlineInfo->nPts = 0; DlineInfo->elm0 = NULL;
                     if (Sfeature->casS2) {
                           new(&Dline->casS2,CasS2);
                           new(&DlineInfo->elm0,MinBoundRect);
                           for (sP = Sfeature->casS2, dP = Dline->casS2; sP;
                                sP=sP->next, dP=dP->next,init = 0) {
                                 DlineInfo->nPts++;
                                 new(&dP->member_DESTINATION_0,Point);
                                 dP->member_DESTINATION_0->x
                                       = sP->element_SOURCE_1->x;
                                 dP->member_DESTINATION_0->y
                                       = sP->element_SOURCE_1->y;
                                 setMinMbr(mnx,x); setMinMbr(mny,y);
                                 setMaxMbr(mxx,x); setMaxMbr(mxy,y);
                                 setNext(sP->next,&dP->next,CasS2); } }
                     break; }
               case 1 : { /* Point */
                     Dfeature->offset = type_DESTINATION_CasS8_casS6;
                     new(&Dpoint,CasS6);
                     Dpoint->id = Sfeature->casS0->id;
                     Dpoint->x = Spoint->x; Dpoint->y = Spoint->y;
                     break; }}
            setNext(s->next,&d->next,Destination); }
}
```

Figure 10.1: An example of a C translator module

A Miranda translator module is generated by a2b from a Miranda translation definition as shown in Figure 10.2. The functions specified within the Miranda translation definition are

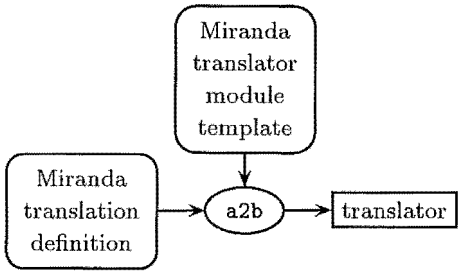


Figure 10.2: Generating a Miranda translator module

inserted into the Miranda translator module template, presented in Appendix F.4.1, to form a complete translator module. Central to a translator module is the function `doTranslation` which, although included within a translator module, is in effect part of the main routine of an interface. This function calls, in order, the following functions: `decodeFile`, to perform the transformation $\mathcal{T}_{mf} \stackrel{decode}{\longrightarrow} \mathcal{T}_{mm}$; `repA2repB`, to perform the transformation $\mathcal{T}_{mm} \stackrel{translate}{\longrightarrow} \mathcal{T}_{mm'}$; and `encodeFile`, to perform the transformation $\mathcal{T}_{mm'} \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf'}$.

The functions `decodeFile`, `encodeFile`, and `doTranslation` are common to all Miranda translator modules and form the Miranda translator module template. Into this template is inserted the function `repA2repB`, and any other functions called from within this function. The function `repA2repB` and any others called by `repA2repB` form the Miranda translation definition, with the function `repA2repB` having the general form described earlier in Section 7.3.1.2.

An example of a generated Miranda translator module is given in Figure 10.3. This module was generated by a2b from the transfer specification comprising the interface and representation definitions given in Figure 7.2 on page 99, and the Miranda translation definition given in Figure 7.10 on page 116.

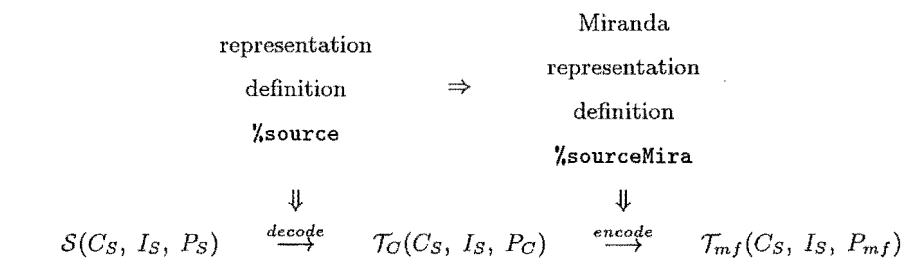
The transformations $\mathcal{T}_C \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf}$ and $\mathcal{T}_{mf'} \stackrel{decode}{\longrightarrow} \mathcal{T}_{C'}$ shown in (10.1) are necessary to transform the data to and from text file representations that can be processed by the functions `decodeFile` ($\mathcal{T}_{mf} \stackrel{decode}{\longrightarrow} \mathcal{T}_{mm}$) and `encodeFile` ($\mathcal{T}_{mm'} \stackrel{encode}{\longrightarrow} \mathcal{T}_{mf'}$). Definitions of these text file representations are generated by a2b from other representations defined in the transfer specification. Generating these definitions is described next.

```
#! /pcks/miranda/mira -exp
doTranslation

%include "/users/cosc/jpp/richard/src/a2b/testfiles/eg1m/sourceMira.m"
%include "/users/cosc/jpp/richard/src/a2b/testfiles/eg1m/destinationMira.m"

decodeFile :: [char] -> [source_source_type]
decodeFile fileName
       = readvals fileName

encodeFile fileName dataString
      = [ Tofile fileName dataString ]

source2destination data
  = map src2destObj data
    where
      src2destObj ((id,0),pts)
        = Destination_choice_1 ((id, # pts, mbr), pts)
          where
            thePts = foldl dopts ((10000, 10000, -10000, -10000), []) pts
            mbr = extract_mbr thePts where extract_mbr (a,b) = a
      src2destObj ((id,1),(x,y):pts)
        = Destination_choice_2 (id, x, y)
      dopts ((mnx, mny, mxe, mxn), xys) (x,y)
          = ((min2 x mnx, min2 y mny, max2 x mxe, max2 y mxn), (x,y):xys)

doTranslation
    = encodeFile outFile (show data)
      where
          outFile = $*!2
          inFile = $*!1
          data = source2destination (hd (decodeFile inFile))
```

Figure 10.3: An example of a Miranda translation module

10.2.1 Generating Miranda representation definitions

Any Miranda representation definition generated by a2b serves the purpose of defining a temporary text file representation for data values during a transfer. Consider the following:



The source data set $\mathcal{S}$, conforming to the representation definition %source, is transformed into a data set $\mathcal{T}_C$ by a decoder module generated by a2b from %source. The data set $\mathcal{T}_C$ is

then transformed, by an encoder module generated by `a2b` from the Miranda representation definition `%sourceMira`, into a data set $\mathcal{T}_{mf}$, conforming to `%sourceMira`. This definition is generated by `a2b` from `%source` which is given in the transfer specification.

After translation, $\mathcal{T}_{mf}(C_S,\ I_S,\ P_{mf}) \xrightarrow{*} \mathcal{T}_{mf'}(C_D,\ I_D,\ P_{mf'})$, the following sequence of transformations occur:

$$\begin{array}{ccccc} & \text{Miranda} & & \text{representation} & \\ & \text{representation} & \Leftarrow & \text{definition} & \\ & \text{definition} & & \texttt{\%destination} & \\ & \texttt{\%destinationMira} & & & \\ & \Downarrow & & \Downarrow & \\ \mathcal{T}_{mf'}(C_D,\ I_D,\ P_{mf'}) & \xrightarrow{decode} & \mathcal{T}_{C'}(C_D,\ I_D,\ P_{C'}) & \xrightarrow{encode} & \mathcal{D}(C_D,\ I_D,\ P_D) \end{array}$$

The data set $\mathcal{T}_{mf'}$, conforming to the Miranda representation definition `%destinationMira`, is transformed into the data set $\mathcal{T}_{C'}$ by a decoder module generated from `%destinationMira`. The Miranda representation definition `%destinationMira` is generated from the representation definition `%destination` which is given in the transfer specification. The data set $\mathcal{T}_{C'}$ is then transformed into the destination data set $\mathcal{D}$, conforming to `%destination`, by an encoder module generated from `%destination`.

Generating encoder and decoder modules from representation definitions was described in Chapter 9. An overview of generating Miranda representation definitions is shown in Figure 10.4 and is now described.

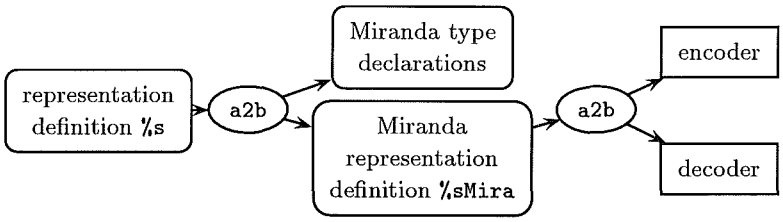


Figure 10.4: Generating a Miranda representation definition `%sMira` from a representation definition `%s` given within a transfer specification

A Miranda representation definition generated by `a2b` comprises type definitions which specify different kinds of data representations in terms of the type constructs provided by the Miranda programming language such as lists, tuples, and algebraic data types. Miranda data values comprising a list are represented within a text file as collections of values, each collection enclosed within the symbols '`[`' and '`]`'. Members of this collection are separated by the symbol '`,`'. A list comprising the integer values `132`, `932`, and `401`, for example, has a text file representation of:

```
[132,932,401]
```

and is specified in a Miranda representation definition as follows:

```
seqOfIntegers : "[" (value ||  ",")* "]" ;
value : integer;
```

Miranda data values comprising tuples are represented within a text file as collections of values, each collection enclosed within the symbols '(' and ')'. Members of the collection are separated by the symbol ','. Consider a tuple comprising the string identifier `"line 132"`, and the list of points `(132.6, 654.87)`, `(423.76, 883.04)`, and `(324.55, 723.15)` where each point is also represented as a tuple comprising two numeric values. This value has a text file representation of:

```
("line 132",[(132.6, 654.87),(423.76, 883.04),(324.55, 723.15)])
```

and is specified in a Miranda representation definition as follows:

```
aLineValue : "(" ( stringIdentifier "," points ) ")" ;
points : "[" ( aPoint || ",")* "]" ;
aPoint : "(" ( x "," y ) ")" ;
stringIdentifier : string;
x,y : real;
```

To illustrate the process of generating Miranda representation definitions, the Miranda representation definitions `%sourceMira` and `%destinationMira` are given in Figure 10.5 for the representation definitions `%source` and `%destination` defined in the transfer specification given in Figure 7.2 on page 99. Also shown for each of these representation definitions is a text file in which the data values shown in Table 7.1 on page 98 are represented according to the Miranda definition.

As well as generating a text file representation, `a2b` also generates a collection of Miranda type declarations, as shown in Figure 10.4. These type declarations define a memory-resident representation for data to be processed by a Miranda translator module.

Each A2B type construct that may be used in a representation definition has a corresponding Miranda type declaration using some combinations of list and tuple data types. The A2B type constructs `{aType}*` and `(aType)*` for defining sets or sequences of one type of data value are represented using Miranda lists. The A2B type constructs `{type1,..,typeN}` and `(type1,..,typeN)` for defining sets or sequences comprising values of different types are represented using tuples.

An A2B choice construct for choosing between different types of data values is represented using a Miranda algebraic type declaration, with each type that may be chosen being represented within the algebraic type using a unique constructor. For example, the representation

Representation definition	Example text-file
%sourceMira sourceMira : "[" (feature || ",")* "]" ; feature : "(" ("(" (id "," type) ")" "," "[" (point || ",")* "]") ")" ; point : "(" (x "," y) ")" ; type, id, x, y: integer ;	[((874,0),[(867,543),(874,550), (880,564)]), ((875,0),[(880,564),(876,580)]), ((876,1),[(872,550)])]
%destinationMira destinationMira : "[" (< "Destination_choice_1" "(" ("(" (id "," nPts "," minBoundRect) ")" "," "[" { point || "," }* "]") ")" | "Destination_choice_2" "(" (id "," x "," y) ")" > || ","}* "]"; point : "(" (x "," y) ")" ; minBoundRect : "(" (mnx "," mny "," mxx "," mxy) ")"; id, nPts, x, y, mnx, mny, mxx, mxy : integer ;	[Destination_choice_1 ((874,3,(867,543,880,564)), [(867,543),(874,550),(880,564)]), Destination_choice_1 ((875,2,(876,564,880,580)), [(880,564),(876,580)]), Destination_choice_2 (876,872,550)]

Figure 10.5: Miranda representation definitions and example text files for the transfer specified in Figure 7.2

definition shown in Figure 10.6(a) would result in a2b generating the Miranda algebraic type declaration shown in Figure 10.6(b) where Ex_aPolygon, Ex_aLine, and Ex_aPoint are type constructors. Also generated by a2b would be the Miranda representation definition shown in Figure 10.6(c).

10.3 Conclusions

An interface comprising a Miranda translator module was found by the author to be slower than an interface which performs the same translation using a C translator module. The slower performance observed was expected, given the additional transformations $\mathcal{T}_C \xrightarrow{encode} \mathcal{T}_{mf} \xrightarrow{decode} \mathcal{T}_{mm}$ and $\mathcal{T}_{mm'} \xrightarrow{encode} \mathcal{T}_{mf'} \xrightarrow{decode} \mathcal{T}_{C'}$, to be performed by an interface with a Miranda translator module.

Use of either the Miranda programming language or the SQL language by the author for defining translator modules was found to produce more compact modules than equivalent translator modules defined using the C programming language. In Table 10.1, for example, is shown the number of lines of code used to define the same translator using the C, Miranda, and SQL languages.

```
%ex
anObject : < aPolygon
           | aLine
           | aPoint> ;
aPolygon : ... ;
aLine    : ... ;
aPoint   : ... ;
```

(a) A representation definition

```
Ex_choice_type ::=
    Ex_aPolygon ex_aPolygon_type
  | Ex_aLine ex_aLine_type
  | Ex_aPoint ex_aPoint_type
ex_aPolygon_type == ...
ex_aLine_type    == ...
ex_aPoint_type   == ...
```

(b) The Miranda type declaration generated for 10.6(a)

```
%exMira
exMira   : < "Ex_aPolygon" aPolygon
           | "Ex_aLine" aLine
           | "Ex_aPoint" aPoint>;
aPolygon : .. ;
aLine    : .. ;
aPoint   : .. ;
```

(c) A Miranda representation definition generated from 10.6(a)

Figure 10.6: An example of generating Miranda algebraic type declarations

Environment	Number of lines	Figure and page
Miranda	14	Figure 7.10 on page 116
C	38	Figure 7.9 on page 115
SQL	12	Figure 5.14 on page 85

Table 10.1: Comparing the number of lines of code for a translator defined using the C, Miranda, and SQL languages

Automatically generating encoder and decoder modules to transform data to and from a relational environment provided by the Ingres DBMS was found to be much more difficult than constructing the same software for the Miranda translation environment, described in Section 10.2.1. The difficulty in generating encoder and decoder modules to transform data to and from a relational environment was defining (at least) a first normal form relational database, with the sequence attributes necessary to maintain the ordering of tuples where required.

Specifying C and Miranda translator definitions is difficult because they are defined in terms of data structure declarations which are generated by a2b from representation definitions. This is a greater problem when defining C translator modules than when defining

Miranda translator modules because reordering of the type definitions within a representation definition may change the names of the generated C data structure declarations. Although these name changes occur without modifying the representation or structure of the defined types, the C translator definition needs to be updated for these new names.

Miranda translator definitions need to be updated only when changes are made to the representation or structure of the types in a representation definition. Reordering type definitions within a representation definition does not affect Miranda translator definitions. More generally, this dependency between representation and translator definitions highlights the need for a notation, to be discussed in Section 14.4, that allows translations to be expressed in terms of the types specified within the representation definitions that form the transfer specification, rather than the data structure declarations generated from these representation definitions.

Overall, the author preferred the translation environment provided by the Miranda programming language to those environments provided by either the C or SQL languages because more compact and robust translators could be defined and additional software to transform data into the Miranda environment could be automatically generated by `a2b`.

Construction of communicator modules

Chapter 11

A communicator module moves data through a communications network from a source communicating interface on one computer system, to a destination communicating interface executing on another computer system. This transformation is described as:

$$\mathcal{N}_S(C_N,\ I_N,\ P_N,\ L_S) \overset{communicate}{\rightsquigarrow} \mathcal{N}_D(C_N,\ I_N,\ P_N,\ L_D)$$

As was explained in Section 6.2.1, a communicator module must be preceded by a presentation layer encoder, and must be followed by a presentation layer decoder. Thus, to move a data set from a source communicating interface to a destination communicating interface, the required transformations are:

Presentation layer	$\mathcal{T}_S \overset{encode}{\longrightarrow} \mathcal{N}_S$	$\mathcal{N}_D \overset{decode}{\longrightarrow} \mathcal{T}_D$
Layers 1–5	$\mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D$	

Construction of a communicator module described in this Chapter does not result in a simple software routine that performs the transformation $\mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D$. Instead, the source and destination communicating interfaces generated by a2b are structured according to the form of an ISODE application, described earlier in Section 4.5. Communication between the initiator and a responder of such an application corresponds to the data transformation $\mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D$, and is accomplished by using the remote operations service (ISO9072-1 1988) as implemented within ISODE. Section 11.1 is a description of the structure of an application corresponding to a pair of communicating interfaces. Construction of these applications using the software tools and run-time libraries provided by ISODE is described in Section 11.2.

11.1 Structure

Transfer specifications of the form described in Chapter 7 may be used to define transfers that are performed by pairs of communicating interfaces. Present implementation of `a2b` allows the construction of only two communicating interfaces, a source and a destination, from any one transfer specification. Thus, only one communicator module may be used within a transfer specification.

In Section 4.5, an application which moves data through a communications network was divided into an initiator and a responder. The transformations $\mathcal{S} \xrightarrow{*} \mathcal{N}_S$, to be performed by the source communicating interface, are implemented within the responder of an application. The transformations $\mathcal{N}_S \xrightarrow{*} \mathcal{D}$, to be performed by the destination communicating interface, are implemented within the initiator of an application. Figure 11.1 is an extended version of Figure 4.4, and is used to show how the various data transformations performed during a transfer correspond with the communication between the initiator and the responder.

The various tasks performed by an initiator and a responder are described in Section 11.1.1. Routines for performing aspects of these tasks are provided within a collection of ISODE run-time libraries, which are described in Section 11.1.2.

11.1.1 Initiators and responders

An initiator performs the following sequence of tasks:

1. Establish association with a responder;
2. Either:
   - Process a command given on the command line and then terminate association with responder, or
   - Enter an *interaction loop* comprising two tasks: get a command from the user; and perform the command. The interaction loop is terminated by the user entering the `quit` command;

   The alternatives correspond to the two forms of an initiator, described earlier in Section 4.5. The first describes the behaviour of an embedded initiator, while the second describes the behaviour of an interactive initiator.

The first of these tasks — establishing communication between the responder and the initiator — is performed by an association control service element (ACSE of Figure 11.1). The second, performed by a remote operations service element (ROSE of Figure 11.1), is the processing of a command and is accomplished in three parts:

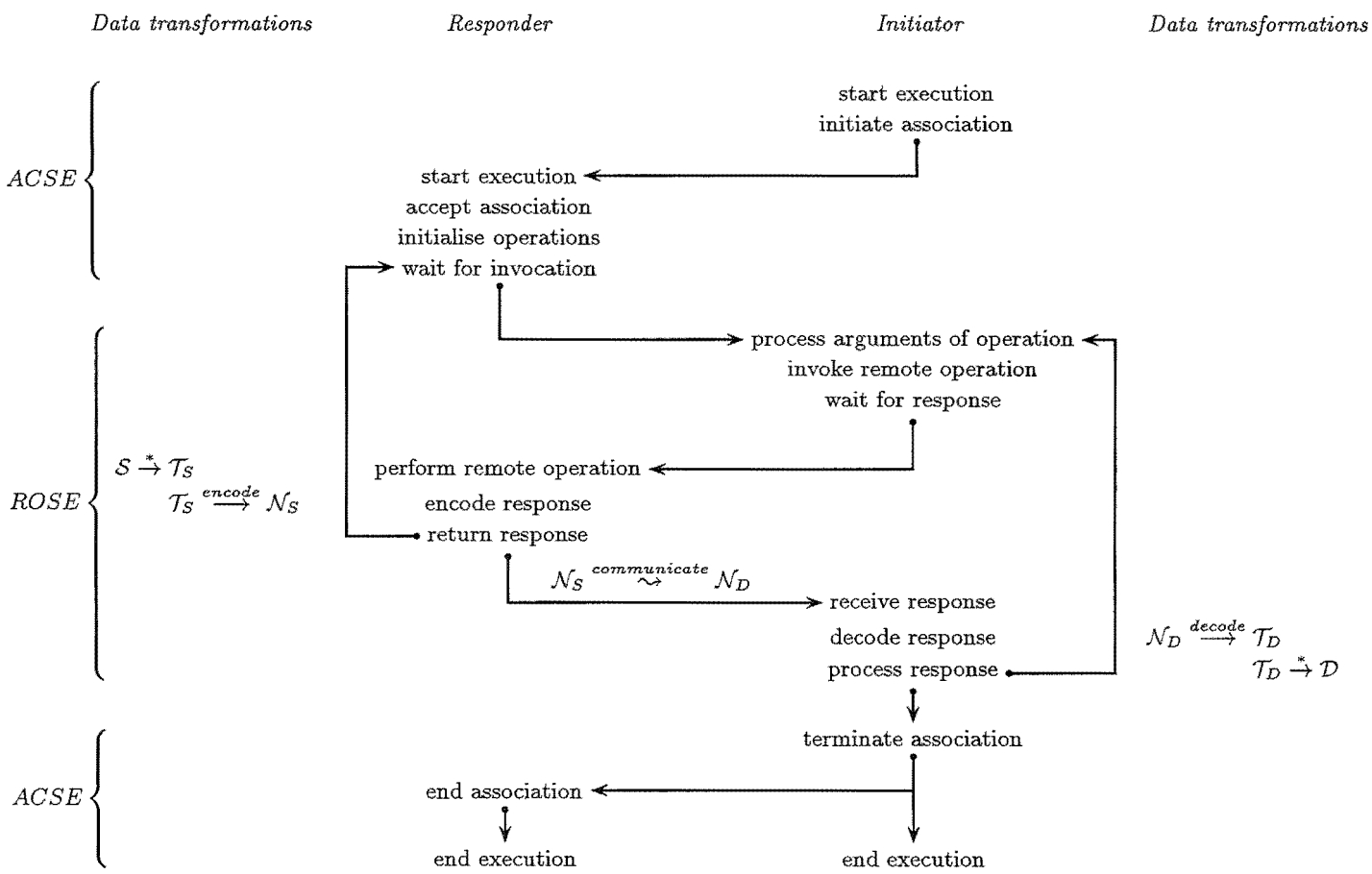


Figure 11.1: Communication between responder and initiator

- Process arguments to be passed to the responder when invoking the remote operation associated with the command;
- Invoke the remote operation. This results in the transformations $\mathcal{S} \xrightarrow{*} \mathcal{N}_S$ of the source communicating interface being performed by the responder, as described below.
- Process the data returned from a successful operation. This processing by the initiator corresponds to performing the transformations $\mathcal{N}_D \xrightarrow{*} \mathcal{D}$ of the destination communicating interface and is divided into two parts:
  - Decode the data values returned by the responder using a presentation layer decoder module. That is, perform the presentation layer transformation $\mathcal{N}_D \xrightarrow{decode} \mathcal{T}_D$; and
  - Perform any further data transformations required to complete the data transfer. That is, apply the application layer transformations $\mathcal{T}_D \xrightarrow{*} \mathcal{D}$.

A responder performs the following sequence of tasks:

1. Form an association with the initiator;
2. Wait for either an invocation of a remote operation, or an indication that the association with the responder is to be terminated;
3. Either:
   - Perform the invoked operation, or
   - Terminate the association with the initiator;

Forming or terminating the association with the initiator is performed by the ACSE. A remote operation is performed by the ROSE, and is accomplished in three parts:

- Execute the remote operation. This corresponds to performing the application layer data transformations $\mathcal{S} \xrightarrow{*} \mathcal{T}_S$;
- Encode the resultant data set for transmission through the network to the initiator. This corresponds to the data transformation, $\mathcal{T}_S \xrightarrow{encode} \mathcal{N}_S$, and is performed by a presentation layer encoder module; and
- Transmit the encoded data set to the initiator.

Tasks common to many initiators and responders are performed by routines contained within run-time libraries provided by ISODE, which are described next.

11.1.2 ISODE run-time libraries

ISODE provides several run-time libraries, and use of the routines within these libraries determine the structure of initiators and responders. Some libraries contain routines which implement aspects of application service elements. For example, the library called `acsap` implements aspects of an association control service element, and `rosap` implements aspects of a remote operations service element. Other ISODE run-time libraries are concerned with more general aspects of the communication process. The `psap` library, for example, contains routines which implement 'presentation syntax abstractions for the machine-independent exchange of data structures' (Rose *et al* 1991, Vol. 1, pg. 115).

To facilitate the construction of applications, ISODE provides two run-time libraries which provide a simplified interface to the routines contained in the `rosap` and `psap` libraries mentioned above. These are:

`rosy` library
: which provides a 'structured interface to the remote operations service' (Rose *et al* 1991, Vol. 4, p. 105), and is to be used in conjunction with the software tool `rosy`;

`pepsy` library
: which provides a table-driven interface to the routines within the `psap` library, and is to be used in conjunction with the software tool `pepsy`.

Some of the more important routines in the `acsap`, `rosy`, and `pepsy` libraries are briefly described below.

11.1.2.1 The `acsap` library

Within the `acsap` library there are routines for managing associations between initiators and responders. These routines are divided into three groups, each dealing with a different aspect of association management. The three are (Rose *et al* 1991, Vol. 2, p. 17):

association establishment
: concerned with connecting the initiator with the responder using the communications network, and initialising both the initiator and the responder;

association release
: concerned with ending the connection between an initiator and a responder; and

association abort
: concerned with abruptly terminating an association, usually because of an error within the initiator or the responder.

The library routine called `AcAssocRequest` is used by the initiator to request an association with a responder. The library routine called `AcAssocResponse` is used by the responder to indicate whether the association is accepted or rejected. The library routine called `AcUAbortRequest` is used by either the initiator or the responder to abort an association.

11.1.2.2 The rosy library

Within the `rosy` library there are routines which manage both the transformation of data to and from the network representation used by ISODE, and the invocation and execution of remote operations. Transforming data to and from the network representation is achieved by the `rosy` routines calling other routines contained within the `pepsy` library, to be described in the next Section. Invocation and execution of remote operations is achieved by the `rosy` routines calling other routines contained within either the `rosap` library, the responder, or the initiator. Examples of the routines provided within the `rosy` library are now described, and routines provided within an initiator and a responder are described in Section 11.2.3

The `RyStub` routine (Rose *et al* 1991, Vol. 4, p. 111) is used by initiators to invoke an operation to be performed by a responder. When a response to this invocation is received, the `RyStub` routine calls a routine provided within the initiator for processing the data returned by the remote operation. The `RyWait` routine (Rose *et al* 1991, Vol. 4, p. 118) is used by responders to wait for an invocation of a remote operation. When an operation is invoked by an initiator, the `RyWait` routine calls the routine within the responder which performs the operation, and then exits.

Either of the routines `RyDsResult`, `RyDsError`, or `RyDsUReject` (Rose *et al* 1991, Vol. 4, p. 116–117) is used by the responder to send the result of an operation to the initiator. The routine `RyDsResult` is to be used when the operation is successfully executed and there are data values to be returned. The routine `RyDsError` is to be used when execution of the operation fails. The routine `RyDsUReject` is to be used when invocation of the operation is rejected.

11.1.2.3 The pepsy library

Within the `pepsy` library are routines which, according to the content of tables processed by these routines, control the transformation of data values between a memory-resident representation derived from an abstract syntax, and a memory-resident representation of the same values in terms of presentation elements. A *presentation element* is 'an object which is used to represent a data structure in a machine independent form' (Rose *et al* 1991, Vol. 1, p. 124). The exact method used to implement presentation elements is unimportant, because

presentation elements are manipulated only by calling routines in the pepsy library.

The routines in the pepsy library perform the presentation layer transformations $\mathcal{T}_S \stackrel{encode}{\longrightarrow} \mathcal{N}_S$ and $\mathcal{N}_D \stackrel{decode}{\longrightarrow} \mathcal{T}_D$. The data sets $\mathcal{T}_S$ and $\mathcal{T}_D$ comprise values conforming to a representation derived from an abstract syntax, and the data sets $\mathcal{N}_S$ and $\mathcal{N}_D$ comprise values represented as presentation elements. The routines in the pepsy library may therefore be regarded as presentation layer encoder and decoder modules which are used by communicating interfaces to transform data to and from the network representation.

Use of the ISODE run-time libraries and the software tools rosy and pepsy to construct applications corresponding to pairs of communicating interfaces is described next.

11.2 Construction

When a2b processes a transfer specification which includes a communicator module, a2b generates an initiator and a responder for performing the data transfer. Generating pairs of initiators and responders is summarised in Figure 11.2 and described below in three parts. Section 11.2.1 is a description of the templates that form the basis of an initiator and a responder. Section 11.2.2 is a description of the remote operations module generated by a2b to define the remote operation, the error codes, and the types of data values processed using this operation. The third part, Section 11.2.3 is a description of generating the software which performs the remote operation.

11.2.1 Initiator and responder templates

The software tool a2b uses templates as the basis for constructing initiators and responders to perform data transfers. The initiator template is presented in Appendix F.2.1 and the responder template is presented in Appendix F.2.2. The templates are incomplete C programs which are based on the initiator and responder programs of the imisc application.

The imisc application is distributed as part of ISODE (versions 7 and 8) (Rose *et al* 1991) for testing the installation of ISODE, and for illustrating the use of the remote operations service implemented as part of ISODE. The imisc application connects to a remote computer system and can be used to retrieve a list of users logged on to this remote computer system, to send a message to any one of these users, and to retrieve the current time.

The initiator and responder templates used by a2b were constructed by the author in two stages. First, the author gained experience with ISODE by constructing an application, called gds/gdc, which had the same structure as the imisc application. The application

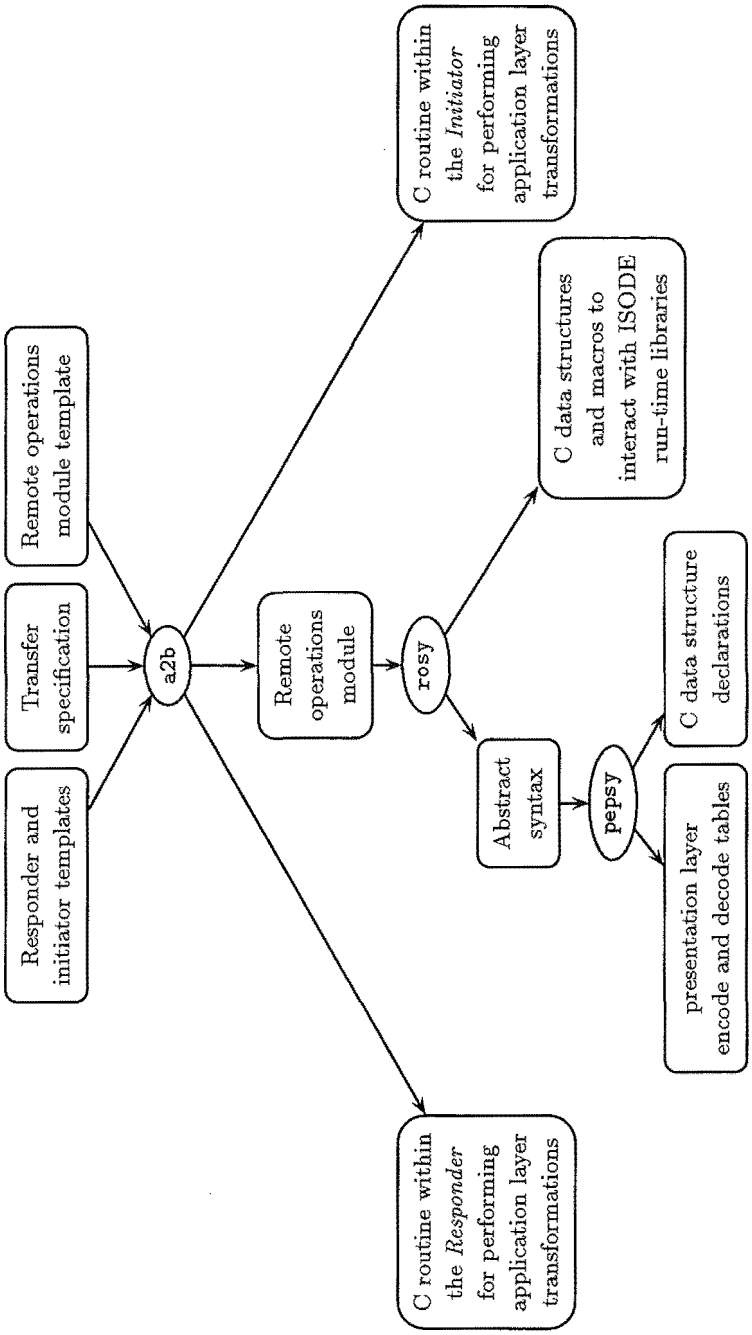


Figure 11.2: Constructing responders and initiators

gds/gdc, to be described in Chapter 12, performed a wide variety of data transfers as described in Section 4.1. Second, the author constructed the templates by simplifying the initiator and responder of the gds/gdc application and removing those parts specific to a data transfer. These parts are to be generated and inserted into the templates by a2b according to the definitions within a transfer specification. For example, a2b generated, as described earlier in Chapter 8, and inserted the main interface routines for the source and destination communicating interfaces.

Present implementation of a2b uses a simple method for locating points in the templates where the source code is to be inserted. A more sophisticated technique for marking and locating positions in which to insert the source code specific to a data transfer will be discussed in Section 14.2. This technique is similar to that used by the Revision Control System (RCS) (Tichy 1985) for inserting version numbers, dates, and comments into source code.

The author decided to modify the existing imisc application, rather than design a new application to transfer geographical data, for two reasons. First, the initiator and responder forming the imisc application provided general error checking and recovery software that could be utilised by the gds/gdc application. Second, the imisc application could be used to demonstrate either the interactive or embedded forms of an initiator. Use of the same structure for the gds/gdc application allowed both forms of an initiator to be investigated.

The initiator and responder templates could be developed further, as will be discussed in Section 14.2, to allow the use of a2b for constructing applications which perform other functions, such as those discussed in Section 4.2. Improvement of the templates is a worthwhile objective, given that these templates will be used each time a2b generates an application for transferring data.

11.2.2 Remote operations module

Within ISODE, a *remote operations module* defines different aspects of the remote operations that form the basis of communication between an initiator and a responder. As was shown in Figure 11.2, this formal definition, generated by a2b from a transfer specification, is processed by the software tool rosy to produce various C data structure and macro definitions. These definitions are used by the initiator and responder to interact with the ISODE run-time libraries, discussed earlier in Section 11.1.2.

The general form of a remote operations module is defined by Rose *et al* (1991, Vol. 4, Chapter 4), and an example of a remote operations module and the transfer specification from which this module would be generated is given in Figure 11.3.

A remote operations module comprises three parts: definition of the remote operations;

```
%a2b src via net to dest
%src
...

%net
net : ( pointId x y );
pointId, x, y : integer;

%dest
...
```

(a) A transfer specification defining a pair of communicating interfaces

```
NET DEFINITIONS ::=
BEGIN
--      Operation Definitions
--
getFile   OPERATION
          ARGUMENT        IA5List
          RESULT          Net
          ERRORS          { unableToOpenFile }
          ::=             0
--
--      Error Definitions
--
unableToOpenFile
          ERROR
          PARAMETER       IA5List
          ::=             0
--
--      Abstract Syntax
--
Net ::= [  APPLICATION 1 ] SEQUENCE {
          pointId [ 0 ]   INTEGER,
          x       [ 1 ]   INTEGER,
          y       [ 2 ]   INTEGER
}
IA5List ::=
        SEQUENCE OF IA5String
END
```

(b) The remote operations module generated by a2b from 11.3(a)

Figure 11.3: An example of a remote operations module, and the transfer specification from which this module is generated by a2b

definition of error codes; and the definition of the types of data values associated with the remote operation.

Definition of a remote operation within a remote operations module is described using BNF notation as follows:

⟨operation definition⟩ ⟶ ⟨name⟩ OPERATION
[ARGUMENT ⟨argument data type⟩]
[RESULT ⟨result data type⟩]
[ERRORS ⟨error names⟩]
::= ⟨operation code⟩

where the mandatory components of an operation's definition are: ⟨name⟩, the name of the operation being defined; and ⟨operation code⟩, an integer value which uniquely identifies the operation from other operations defined in the module. The optional components of an operation's definition are: ⟨argument data type⟩, the type of the data value that is sent as an argument to the operation; ⟨result data type⟩, the type of the data value that is to be returned on successfully performing the operation; and ⟨error names⟩, the errors that may be returned when an operation is unsuccessful.

The software tool `a2b` generates a remote operations module using a template that contains the definition of a remote operation, called `getFile`. This operation forms the basis for communication between an initiator and a responder generated by `a2b`. Definition of the operation `getFile` is shown in Figure 11.3.

Definition of the error codes within a remote operations module has a form similar to that of remote operation definitions. Using BNF, the form of an error message definition is:

⟨operation definition⟩ ⟶ ⟨error name⟩ `ERROR`
[`PARAMETER` ⟨parameter data type⟩]
: : = ⟨error code⟩

where ⟨error name⟩ is the name of the error being defined and is used in the ⟨error names⟩ component of a remote operations definition, and ⟨error code⟩ is an integer value which uniquely identifies the error from other errors defined in the module. The single optional component, ⟨parameter data type⟩, is the type of a data value that may be returned by the responder to provide additional information describing the nature of the error. This data value is returned, together with the error code, when the error occurs.

There are 8 errors defined within the remote operations module template used by `a2b`. Only one of these errors is shown in Figure 11.3, and will be returned by the responder when the routine performing this operation fails to open a file containing data to be decoded. This error is defined with a parameter that will be used by software generated by a future version of `a2b` to explain why the file could not be opened. One possible reason is that the responder was not authorised to access this file.

The third part of a remote operations module, an abstract syntax, defines the types of data values sent between the initiator and responder. Examples of these values are: an argument to the remote operation; the result of a successful remote operation; and the parameter describing an error. An abstract syntax is defined within a remote operations module using Abstract Syntax Notation.1 (ASN.1) (ISO8824 1987), to be summarised in Appendix A.2.

The abstract syntax which defines the type of the data value to be returned by the remote operation is generated by a2b from the representation definition having the name given after the keyword via in the interface definition of a transfer specification. For example, the data type called Net is defined by the abstract syntax which forms a part of the remote operations module shown in Figure 11.3(b). This abstract syntax was generated from the representation definition called %net shown in Figure 11.3(a).

In the definition of getFile generated by a2b, the data value to be returned by getFile is defined to be of the type called Net. Note also that the name of the remote operations module, given before the keyword DEFINITION in the first line of the remote operations module shown in Figure 11.3(b), is taken from the name of the representation definition from which this module was generated.

Definition of the types of data values given either as an argument to a remote operation or as a parameter to an error, are included within the remote operations template used by a2b because they are common to all modules generated by a2b.

11.2.3 Software to perform the remote operation

To perform a remote operation, the following is needed:

- a routine within the initiator to process the arguments to be sent with the invocation of a remote operation;
- a routine within the responder to perform the remote operation. In terms of the data transfer, this routine performs the sequence of application layer data transformations $\mathcal{S} \xrightarrow{*} \mathcal{T}_S$;
- tables used by the routines within the ISODE run-time library pepsy, discussed in Section 11.1.2.3. These routines perform, what in this thesis have been termed, the presentation layer encoding and decoding data transformations $\mathcal{T}_S \xrightarrow{encode} \mathcal{N}_S$ and $\mathcal{N}_D \xrightarrow{decode} \mathcal{T}_D$;
- a routine within the initiator to process the data values returned by the remote operation. In terms of the data transfer, this routine performs the sequence of application layer data transformations $\mathcal{T}_D \xrightarrow{*} \mathcal{D}$.

Construction of this software is now described.

11.2.3.1 Processing arguments for a remote operation

Before invoking an operation, a routine within the initiator processes any arguments to be sent with the invocation of a remote operation. In the case of the initiators generated by a2b,

this routine is called `do_getFile` and transforms the name of the data file to be transferred into the C data structure representation required by the routines in the `pepsy` library.

11.2.3.2 The remote operation

Before discussing the C routine within the responder that performs the remote operation, a description is given below of a C data structure which is used to link this routine to the operation code given in the remote operations module, discussed earlier in Section 11.2.2.

The C data structure comprises an array of `struct`, each `struct` containing three values which describes an operation. These are:

- an operation's name;
- an integer value uniquely identifying the operation. This value must be the same as the value of the operation's code given in the remote operations module; and
- the name of the C routine which performs the operation.

An example is shown in Figure 11.4, and was generated by `a2b` from the transfer specification given in Figure 11.3(a).

```
struct dispatch {
    char   *ds_name;
    int     ds_operation;
    IFP     ds_vector;
};
static struct dispatch dispatches[] = {
    "getFile", operation_NET_getFile, interfaceBody,
    NULL
};
```

Figure 11.4: C data structure declaration generated by `a2b` and inserted into a responder template

In Figure 11.4, the remote operation's name is "`getfile`" and will be the same for any instance of this C data structure generated by `a2b`. The term `operation_NET_getFile` is a C constant generated by `rosy` and corresponds to the remote operation's code, a unique integer value defined within the remote operations module. This C constant would be defined within the source code of the responder as follows:

```
#define operation_NET_getFile 0
```

The C routine which performs the application layer transformations $\mathcal{S} \xrightarrow{*} \mathcal{T}_S$ must have the same name, `interfaceBody`, as that given in the C data `struct`, shown in Figure 11.4. This name will be the same for any C routine generated by `a2b` to perform these application

layer transformations. The general form of this routine is shown in Figure 11.5, and different parts of this routine are discussed below.

```
static int interfaceBody(int sd, struct RyOperation *ryo,
                         struct RoSAPinvoke *rox,
                         caddr_t in, struct RoSAPindication *roi)
{
      int result;
      char *datafile, *tmpPath, *s, *tmpFile();
      FILE *fd;

      if ((result =doPreamble(sd, ryo, rox,
                              (struct type_A2BComModName_IA5List  *) in,
                              roi, &datafile)) == NOTOK)
            goto out;

@% debugging loop - if requested

      tmpPath = (char *)malloc(strlen(datafile));
      strcpy(tmpPath, datafile);
      for (s = tmpPath+strlen(tmpPath); *s!= '/'; s--) ;
      *(s+1) = '\0';

@% the main interface routine generated by a2b

      result = OK;

   out:;
      free(datafile);

      return result;

}
```

Figure 11.5: General form of routine for performing remote operation

The five arguments to this routine have a set form defined by Rose *et al* (1991, Vol. 4, Section 8.6.1). Briefly, these arguments are:

`sd` a description of the association between the initiator and the responder;

`ryo` a structure containing pointers to presentation layer encoder and decoder modules for transforming data to and from the network data representation;

`rox` a 'structure containing miscellaneous information regarding the operation being invoked' (*op cit*);

`in` a pointer to the operation's argument, if any; and

`roi` a structure in which the error code, if any, is returned.

The flag `@% debugging loop - if requested` will be removed by `a2b` and an infinite loop may be inserted if requested by the person using `a2b`. This loop is:

```
result = 1;
while (result)
    ;
```

While the responder executes this infinite loop, a person constructing the responder can attach to the executing process by using a source level debugger, exit the infinite loop by assigning the integer value 0 to the variable `result`, and then trace the execution of the routine corresponding to the source communicating interface. The flag:

```
@% the main interface routine generated by a2b
```

is replaced with the main routine of the source communicating interface, which is generated as was described in Chapter 8. This main interface routine comprises calls to some combination of application layer decoder, encoder, and translator interface modules to perform the sequence of transformations $\mathcal{S} \xrightarrow{*} \mathcal{T}_S$. The main interface routine is concluded by a call to a routine called `sendData`. Only one argument is given to this routine, the data values produced by the remote operation that are to be returned to the initiator.

The routine `sendData` performs the data transformations occurring at or below the presentation layers of the ISO reference model, that is, the transformations described as:

$$\begin{array}{ll} \text{Presentation layer} & \mathcal{T}_S \overset{encode}{\longrightarrow} \mathcal{N}_S \\ \text{Layers 1–5} & \mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D \end{array}$$

In practice, the routine `sendData` uses the `rosy` library routine called `RyDsResult` to perform these transformations. The `RyDsResult` routine uses other routines in the `pepsy` library, as was discussed earlier in Section 11.1.2.2, for performing the presentation layer transformation $\mathcal{T}_S \overset{encode}{\longrightarrow} \mathcal{N}_S$, and routines in the `rosap` library for performing the transformation $\mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D$.

Presentation layer transformations are specific to the types of values being transformed. Routines in the `pepsy` library are table-driven, allowing the same routines to perform a variety of transformations by using different tables, each determining a different transformation. Construction of these tables is discussed below.

11.2.3.3 Tables governing the presentation layer transformations

The presentation layer transformations $\mathcal{T}_S \overset{encode}{\longrightarrow} \mathcal{N}_S$ and $\mathcal{N}_D \overset{decode}{\longrightarrow} \mathcal{T}_D$ are performed by routines in the `pepsy` run-time library according to tables generated by the software tool `pepsy` from an abstract syntax. As shown in Figure 11.2, this abstract syntax is extracted

by `rosy` from the remote operations module which is itself generated by `a2b` from a transfer specification.

The data set $\mathcal{T}_S$, to be transformed by the routines in the `pepsy` run-time library, must be represented according to the C data structure declarations which were generated by `pepsy`, together with the tables which are used by the `pepsy` routines. Similarly, the data set $\mathcal{T}_D$ produced by the `pepsy` routines will be represented according to C data structure declarations generated by `pepsy`.

These tables, together with the routines provided within the run-time library `pepsy`, constitute the presentation layer decoder and encoder modules defined earlier in Sections 6.1.1 and 6.1.2 respectively.

11.2.3.4 Processing the data values returned by the remote operation

The initiator processes the data returned by the execution of a remote operation using a routine which performs the application layer transformations of the destination communicating interface. The routine and the remote operation that produces the data to be processed by this routine form a *command* which is defined in a table within an initiator. The C data structure declaration and initialisation of this table, shown in Figure 11.6, is the same for all initiators generated by `a2b`.

Three commands are defined: `getFile`, which results in the invocation of the remote operation to transfer the data; `help`, which results in the initiator displaying a short description of these commands; and `quit`, which results in the user initiating the termination of the association between the initiator and the responder.

As can be seen by the declaration of the C data structure `dispatch` in Figure 11.6, an entry in the table `dispatches` comprises:

`ds_name`
: the name of a command;

`ds_operation`
: the code of the remote operation. These integer values correspond to those defined within the remote operations module and the C data structure defined within the source code of the responder;

`ds_argument`
: a pointer to a routine for processing the argument of an operation before invoking the operation with this argument. This routine was discussed in Section 11.2.3.1;

```
struct dispatch {
      char   *ds_name;
      int    ds_operation;
      IFP    ds_argument;

      modtyp *ds_fr_mod;        /* pointer to table for argument type */
      int     ds_fr_index;      /* index to entry in tables */

      IFP     ds_result, ds_error;
      char   *ds_help;
};
static struct dispatch dispatches[] = {

    "getFile",  operation_DESTINATION_getFile,
    do_getFile, &_ZDESTINATION_mod, _ZIA5ListDESTINATION,
    interfaceBody, error,
    "SYNOPSIS\n\tgetFile srcFilename destFilename\n\n\
     DESCRIPTION\n\tTransfer the named source datafile to the named\
                   destination datafile\n",

    "help", 0,
    do_help, NULL, 0,
    NULLIFP, NULLIFP,
    "print this information",

    "quit", 0,
    do_quit, NULL, 0,
    NULLIFP, NULLIFP,
    "terminate the association and exit",

    NULL
};
```

Figure 11.6: Definition of table which specifies pairs of remote operations and destination communicating interfaces

`ds_fr_mod, and ds_fr_index`

miscellaneous values required for interaction with routines provided within the ISODE run-time libraries;

`ds_result`

a pointer to a routine within the initiator for performing the application layer transformations $\mathcal{T}_D \xrightarrow{*} \mathcal{D}$, where $\mathcal{T}_D$ is the data set returned by the responder in response to performing a remote operation;

`ds_error`

a pointer to a routine for processing the error returned when a remote operation fails. This routine is provided within the initiator template;

`ds_help`

a string value containing a brief description of the command.

As can be seen from the entry describing the `getFile` command, `interfaceBody` is the name of the routine which performs the application layer transformations $\mathcal{T}_D \xrightarrow{*} \mathcal{D}$. The general form of this routine is shown in Figure 11.7, and parts of this routine are described below.

```
static int interfaceBody (int sd, int id, int dummy,
                          struct type_A2BComModName_A2BCmn *A2Bcmn,
                          struct RoSAPindication *roi)
{
      FILE *fd;

@% the main interface routine generated by a2b

return OK; }
```

Figure 11.7: General form of routine corresponding to the destination communicating interface

The five arguments to this routine have a set form defined by Rose *et al* (1991, Vol 4., Section 8.4). Briefly, these arguments are:

`sd`

a description of the association between the initiator and the responder;

`id`

the invocation identifier associated with the result or error. Essentially, this value indicates which execution of a remote operation generated the data to be processed by this routine;

`dummy`

an indication of whether the execution of the remote operation was successful, unsuccessful, or rejected;

`A2Bcmn`

the data to be processed by this routine. The declaration of this argument is replaced with a declaration using the appropriate name of the C data structure generated for the expected data value. For example, given the transfer specification in Figure 11.3(a), the data value contained within this argument would conform to the representation definition `%net`. Therefore, the declaration for this argument would be:

```
struct type_NET_Net *netData,
```

`roi`

a structure in which, amongst other things, the error code, if any, is returned.

The flag `@% the main interface routine generated by a2b` is replaced with the main routine of the source communicating interface, which is generated as was described in Section 8.

The use of ISODE and the general structure of the applications generated by `a2b` are based on the author's experience of constructing an application called `gds/gdc`, which is discussed in the next Chapter.

The gds/gdc application

Chapter 12

The concept of a communicating interface and some practical examples of this type of interface were discussed together in Chapter 4. As was explained in Chapter 11, the concepts used within a2b for constructing pairs of communicating interfaces were derived from the author's experience of constructing the gds/gdc application by hand within ISODE. The structure of this application and the communicating interfaces that form this application are now described.

12.1 Structure

The application gds/gdc (geographical data server / geographical data client) has the general structure shown in Figure 12.1. This structure was discussed in Section 4.5, and comprises an initiator and a responder which communicate with each other using the remote operations service.

There are two significant differences between the structure of the gds/gdc application, and the structure of the applications now generated by a2b. First, the gds/gdc application comprises many different pairs of source and destination communicating interfaces, as explained earlier in Section 4.1. Each operation implemented within the responder of the gds/gdc application will use whichever source communicating interface is necessary to transform the desired data set from the text file representation to which this set conforms, into the memory-resident representation common to all communicating interfaces (the data sets $\mathcal{T}_S$ in Figure 12.1). Thus the responder, rather than the initiator, decides which source communicating interface to use.

In contrast, the applications discussed earlier in Section 11.1 are designed for a single pair of communicating interfaces, a source communicating interface implemented within the

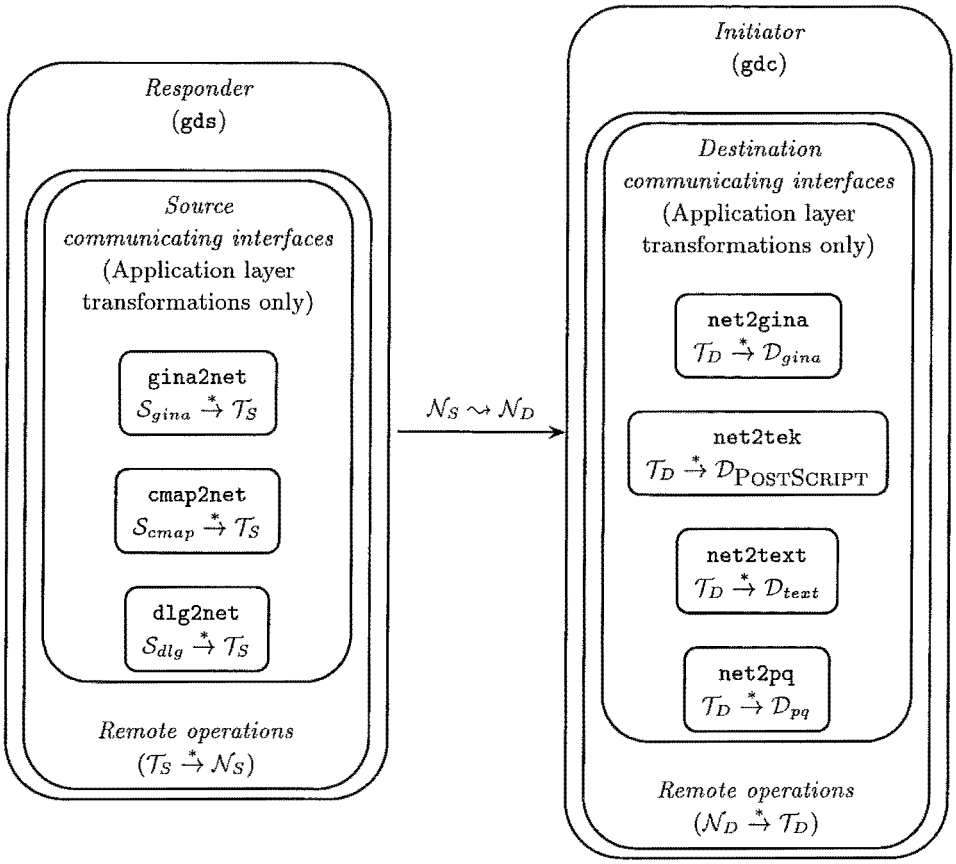


Figure 12.1: Structure of the gds/gdc application

responder and a destination communicating interface implemented within the initiator. Consequently, data sets conforming to only one text file representation can be transferred using the generated initiator and responder.

The second difference is that the responder of the gds/gdc application provides a primitive data directory in which is described the spatial domain of each data set accessible to the communicating interfaces within the responder. This data directory facility, described below, is not provided within the applications discussed earlier in Section 11.1

The initiator (gdc) and responder (gds), forming the application gds/gdc, execute on different Sun work-stations. They communicate with each other using an Ethernet communication network and the communication protocols provided by ISODE (Section 4.5). As was explained in Section 11.2.1, the initiator and responder of the application gds/gdc are based on the initiator and responder of the imisc application which is distributed as part of ISODE.

Data values sent through the network between the initiator and responder of the application `gds/gdc` conform to a network representation that is defined by a conceptual schema, an implementation schema, and a physical schema. The conceptual and implementation schemas of the network representation define a collection of data types which are the same as those defined by the Gina file format (GeoVision 1986). The physical schema to which these data values conform is defined by the ISO 8825 standard, which is described in Appendix A.3.

Throughout this Chapter, sets of data values conforming to the network representation used by the application `gds/gdc` will be described either by $\mathcal{N}_S$, when the data set is located on the source computer system where the responder of the application `gds/gdc` is executing, or by $\mathcal{N}_D$, when the data set is located on the destination computer system where the initiator of the application `gds/gdc` is executing.

A data transfer is performed by the application `gds/gdc` in three parts, each comprising a sequence of steps which are described below. After this general description, key components of these steps will be discussed in more detail for the initiator and the responder.

The first part of a transfer is performed by `gdc`, the initiator of the application `gds/gdc`, and comprises the following steps:

1. Establish a communication link with `gds`, the responder of the application `gds/gdc`;
2. Read a command from the user;
3. Execute a function to process any arguments of the command;
4. Invoke the operation associated with the command to be performed by `gds`.

At this point, the initiator `gdc` waits for a response from the responder `gds`, which performs the second part of the transfer. The second part of the transfer comprises the following steps:

5. Arguments of the operation invoked by the initiator `gdc` are used by the responder `gds` to select data files which may contain data to be transferred. The responder `gds` maintains a directory containing a description of all accessible data files.
6. Each of the selected data files is processed by the source communicating interface appropriate for the representation of data in this file. The various source communicating interfaces that may be used will be discussed in Section 12.2;
7. Each communicating interface produces a data set $\mathcal{T}_S$ which is processed further by the responder `gds` in accordance with the particular operation requested by the initiator `gdc`. This processing selects data values from $\mathcal{T}_S$ that are either within a specified domain, or are of a particular type, and encodes them into the data set $\mathcal{N}_S$, a set of values conforming to the required network representation.

At this point, gds responds to gdc either by sending the data set $\mathcal{N}_S$, or by sending an error message which indicates why the operation was unsuccessful. The initiator gdc then performs the third and final part of the transfer, which comprises the following steps:

8. If the response to the operation is an error, then an error handling routine is executed to inform the user of the error. Otherwise, the data set $\mathcal{N}_D$ produced by the responder gds is decoded from the network representation into the data set $\mathcal{T}_D$, which is passed on to the destination communicating interface associated with the command entered by the user;

9. The destination communicating interface completes the data transfer by transforming the data set $\mathcal{T}_D$ into a data set that conforms to the representation required for the command entered by the user. Destination communicating interfaces will be described in Section 12.3;

10. When gdc executes as an embedded initiator (see Section 4.5) gdc breaks the communication link with the responder gds and terminates. When executing as an interactive initiator, gdc prompts the user to enter another command.

The application gds/gdc is now discussed in more detail, starting with gds because the responder comprises the source communicating interfaces that perform the first of the data transformations constituting any transfer.

12.2 The responder gds

The responder gds provides four remote operations that may be invoked by the initiator. Two of these, `op-GD-getFile` and `op-GD-getDomain`, are for accessing a variety of data sets, each conforming to a representation defined by either the Colourmap (CSIRONET 1986), Gina (GeoVision 1986), or the DLG (Geological Survey 1990) file format. These operations will select whichever source communicating interface is required as a consequence of the representation used by the desired data set. The other two operations, `op-GD-getLines` and `op-GD-getRoads`, are provided for accessing only those data sets conforming to the representation defined by the Gina data file format. Any of these operations is accomplished by the responder gds performing the three Steps numbered 5, 6, and 7 in the description of a data transfer given in Section 12.1.

In Step 5, the arguments of the requested operation are used by the responder gds to select the data files for processing. All data files that can be processed by the responder gds

are listed in a directory. An entry in this directory comprises the name of the data file, and the minimum bounding rectangle of the data values represented within the file.

The operation `op-GD-getFile` has one argument, the name of the file to be transferred, and is used to find this file's entry in the directory. The other operations `op-GD-getDomain`, `op-GD-getLines`, and `op-GD-getRoads` also have one argument, coordinates of the bottom left and upper right hand corners of the minimum bounding rectangle of the domain. Any data file within the directory which has a minimum bounding rectangle that overlaps this domain is selected by the responder `gds` for further processing in Step 6 of Section 12.1.

The application layer transformations of the source communicating interfaces used by the responder `gds` in Step 6 are functions implemented within `gds`. Three source communicating interfaces or functions are implemented:

`cmap2net`

performs the following sequence of transformations:

$$\mathcal{S}_{cmap} \overset{decode}{\longrightarrow} \mathcal{T}_0 \overset{translate}{\longrightarrow} \mathcal{T}_S$$

where

$\mathcal{S}_{cmap}$

is a data set conforming to the Colourmap data file format (CSIRONET 1986);

$\mathcal{T}_0$

is a data set comprising the same values as those within the set $\mathcal{S}_{cmap}$, but which have a memory-resident physical representation; and

$\mathcal{T}_S$

is a data set comprising values which conform to the memory-resident network representation defined for any data values sent between the responder `gds` and initiator `gdc`.

`gina2net`

performs the following sequence of transformations:

$$\mathcal{S}_{gina} \overset{decode}{\longrightarrow} \mathcal{T}_S$$

where

$\mathcal{S}_{gina}$

is a data set conforming to the Gina data file format;

$\mathcal{T}_S$

is a data set as described above.

No translation is performed by this interface because the data sets $\mathcal{S}_{gina}$ and $\mathcal{T}_S$ conform to the same conceptual and implementation schemas defined for the network representation. That is, the sets $\mathcal{S}_{gina}$, $\mathcal{T}_S$, and $\mathcal{N}_S$, all comprise values of the types defined by the Gina data file format;

`dlg2net`

performs the following sequence of transformations:

$$\mathcal{S}_{dlg} \stackrel{decode}{\longrightarrow} \mathcal{T}_0 \stackrel{translate}{\longrightarrow} \mathcal{T}_S$$

where

$\mathcal{S}_{dlg}$

is a data set conforming to the DLG (Digital Line Graph) data file format;

$\mathcal{T}_0$

is a data set comprising the same values as those within the set $\mathcal{S}_{dlg}$, but which have a memory-resident physical representation; and

$\mathcal{T}_S$

is a data set as described above.

After transformation into the data set $\mathcal{T}_S$, Step 7 of Section 12.1 is performed as part of the operations `op-GD-getDomain`, `op-GD-getLines`, and `op-GD-getRoads`. Before this data set is sent by the responder `gds` to the initiator `gdc`, those spatial data values in this set that do not overlap the minimum bounding rectangle supplied as an argument of the operation are removed. Furthermore, in the case of the operation `op-GD-getLines`, any spatial data value that is not a line is removed, and in the case of the operation `op-GD-getRoads` any spatial data value not describing a road boundary is removed. Finally, the set of data values to be returned to the initiator are encoded into the network representation. That is, the transformation $\mathcal{T}_S \stackrel{encode}{\longrightarrow} \mathcal{N}_S$ is performed.

Once all processing of the data values is complete, the responder `gds` sends the data set $\mathcal{N}_S$ to the initiator `gdc`, and waits for `gdc` to either request another operation to be performed, or signal that `gds` should terminate execution.

12.3 The initiator gdc

The initiator gdc may be executed in one of two modes: embedded mode, where the user specifies the command and any arguments when gdc is executed; and interactive mode, where the user provides no command when gdc is executed and, as a consequence, is prompted for commands by gdc.

The initiator gdc provides a user with 13 commands. Examples of these commands are: quit, for terminating an interactive session with this initiator; intro, for viewing a brief description of the application gds/gdc; and help, for viewing an on-line description of the commands provided by gdc. The other 10 commands are listed in Table 12.1 and result in different data transfers being performed by the application gds/gdc.

Each command is mapped by the initiator gdc onto:

- a function to process any arguments given as part of the command. This function performs Step 3 of the transfer process described in Section 12.1, and is the same function for all commands;
- the operation to be performed by the responder gds, as was described earlier in Section 12.2; and
- a function to process the results of a successful operation. This function performs the application layer transformations of a destination communicating interface, and implements Step 9 of the transfer process described in Section 12.1.

Table 12.1 shows the names of the remote operations to be performed by the responder gds and the destination communicating interface to be executed by gdc, for each command entered by a user of the gds/gdc application. A synopsis of the transfer performed by each combination of an operation and a destination communicating interface is also given in Table 12.1. The destination communicating interfaces provided by the initiator gdc are now described.

The four destination communicating interfaces implemented within the initiator gdc perform only the application layer transformations of a transfer. The other transformations are performed by the mechanisms provided by the remote operations service. The four interfaces are:

net2gina
: performs the following sequence of transformations:

$$\mathcal{T}_D \xrightarrow{encode} \mathcal{D}_{gina}$$

Command	Operation	Destination communicating interface	Synopsis
getFile	op-GD-getFile	net2text	Transfer data represented within some text file, into data textually displayed on a screen.
getDomain	op-GD-getDomain	net2text	Transfer data that lie within some domain and are represented within one or more different text files, into data textually displayed on a screen.
showFile	op-GD-getFile	net2tek	Transfer data represented within some text file, into data graphically displayed on a screen.
showDomain	op-GD-getDomain	net2tek	Transfer data that lie within some domain and are represented within one or more different text files, into data graphically displayed on a screen.
showLines	op-GD-getLines	net2tek	Transfer data that lie within some domain, are represented within one or more different text files, and are lines, into data graphically displayed on a screen.
showRoads	op-GD-getRoads	net2tek	Transfer data that lie within some domain, are represented within one or more different text files, and describe road boundaries, into data graphically displayed on a screen.
ginaFile	op-GD-getFile	net2gina	Transfer data represented within some text file, into data represented within a Gina formatted text file.
ginaDomain	op-GD-getDomain	net2gina	Transfer data that lie within some domain and are represented within one or more different text files, into data represented within a Gina formatted text file.
pqFile	op-GD-getFile	net2pq	Transfer data represented within some text file, into data represented within a script for insertion into a postgres database.
pqDomain	op-GD-getDomain	net2pq	Transfer data that lie within some domain and are represented within one or more different text files, into data represented within a script for insertion into a postgres database.

Table 12.1: The user commands provided by the initiator gdc

where

$\mathcal{T}_D$

is a data set conforming to the memory-resident network representation used for any data values sent between `gds` and `gdc`; and

$\mathcal{D}_{gina}$

is a data set conforming to the Gina data file format.

As with the interface `gina2net` described earlier in Section 12.2, no translation is performed by the interface `net2gina`;

`net2tek`

performs the following sequence of transformations:

$$\mathcal{T}_D \stackrel{translate}{\longrightarrow} \mathcal{T}_G \stackrel{translate}{\longrightarrow} \mathcal{T}_1 \stackrel{encode}{\longrightarrow} \mathcal{D}_{POSTSCRIPT}$$

where

$\mathcal{T}_D$

is a data set as described above;

$\mathcal{T}_G$

is a data set comprising data values represented as arguments to graphics commands for plotting the values on a variety of devices;

$\mathcal{T}_1$

is a data set comprising data values the same as those within the set $\mathcal{T}_G$, but the commands are specific to POSTSCRIPT (Adobe 1985); and

$\mathcal{D}_{POSTSCRIPT}$

is a data set comprising data values the same as those within the set $\mathcal{T}_1$, but which have a text file representation suitable for sending to POSTSCRIPT output devices;

`net2pq`

performs the following sequence of transformations:

$$\mathcal{T}_D \stackrel{encode}{\longrightarrow} \mathcal{D}_{postquel}$$

where

$\mathcal{T}_D$

is a data set as described above; and

$\mathcal{D}_{postquel}$

is a data set comprising data values which are represented within a postquel script for inserting these values into a `postgres` database;

`net2text`

performs the following sequence of transformations:

$$\mathcal{T}_D \stackrel{encode}{\longrightarrow} \mathcal{D}_{text}$$

where

$\mathcal{T}_D$

is a data set as described above; and

$\mathcal{D}_{text}$

is a data set comprising data values the same as those within the set $\mathcal{T}_D$ but which have a text file representation defined by ISODE for data conforming to the physical representation defined by the ISO 8825 standard for data transmission through a communications network.

12.4 Examples of data transfers

The data sets shown in Figures 12.2 to 12.6 were created by combining the destination communicating interface called `net2tek` with the source communicating interfaces provided by `gds`. In this way, the application `gds/gdc` could transfer a variety of data sets stored on one computer system for graphical display on some other. In Figure 12.2 a collection of property boundaries is shown, and in Figure 12.3 a collection of road boundaries is shown. Both collections were part of the Dosli data set described in Section 13.1. Other examples of the data sets made accessible using the communicating interfaces within `gds` and `gdc` included:

- a data set represented according to the DLG data file format (Geological Survey 1990). This data set comprised a map shown in Figure 12.4 which describes the land use capability of Banks Peninsula based upon the rock type, soil, slope, erosion, and vegetation; and

- a variety of data sets represented according to the Colourmap file format (CSIRONET 1986). Some of these data sets comprised maps such as the one shown in Figure 12.5 which described country boundaries. Other data sets comprised maps such as the

Figure 12.2: A plot of some cadastral data values conforming to the Gina data file format

one shown in Figure 12.6 which described statistical, local government, and postal boundaries for parts of Australia.

12.5 Conclusions

In constructing the `gds/gdc` application, many useful concepts were learnt by the author and used in the development of the software tool `a2b`. These concepts and techniques are described below.

The author's immediate objective for `a2b` was to use this tool for constructing interfaces that transferred data from one representation on one computer system to another representation on a different computer system. Consequently, a responder need have only one source communicating interface which is used by any operations provided by this responder. This is less complicated than the `gds/gdc` application, where one operation could use one of a variety of communicating interfaces according to the representation of the data set to be transferred.

The author also realised the potential for constructing communicating interfaces that could be used in a wide variety of ways. Some interfaces may be used to transfer entire data

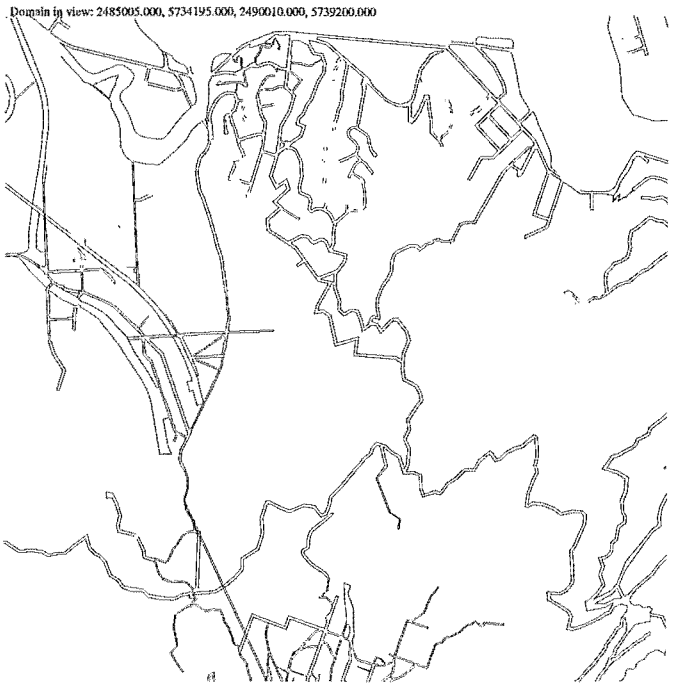


Figure 12.3: A plot of some road data values conforming to the Gina data file format

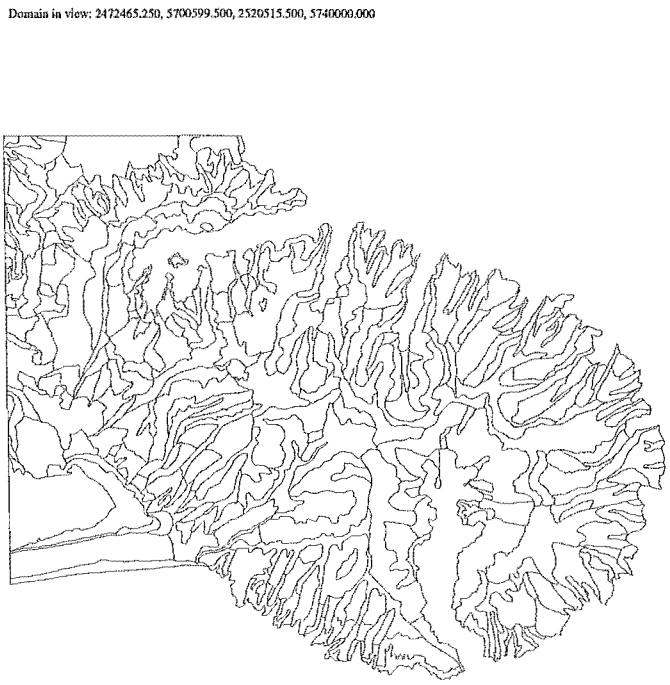


Figure 12.4: A plot of some data values for Banks Peninsula conforming to the DLG data file format

Figure 12.5: A plot of some data values for the Far East conforming to the Colourmap data file format

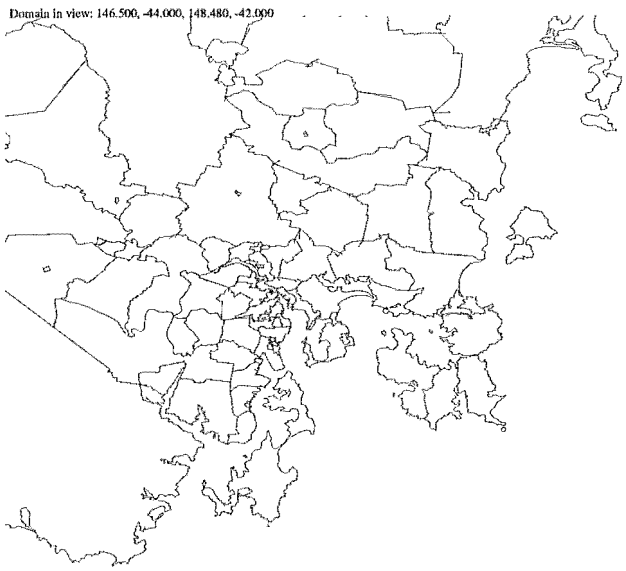


Figure 12.6: A plot of some data values for the 1981 postal zones of Hobart in Tasmania, conforming to the Colourmap data file format

files, others may be used to transfer a data set created from many different data files. To allow for a variety of interfaces to be constructed, templates were used by `a2b` as the basis for generating interfaces. As will be explained in Section 14.2, future versions of the author's software tool will be able to construct a variety of interfaces by using different templates.

Source code for the `imisc` application served as a useful platform for constructing the `gds/gdc` application, and later for the applications generated by `a2b`. This source code assisted the author to identify parts of the ISODE remote operations service that could perform the presentation layer data transformations, and those parts of the remote operations service that could use other routines generated by `a2b` to perform the application layer data transformations necessary to accomplish the transfer. In hindsight, however, the existing templates based on the `imisc` application are excessively complicated for the transfers performed by these interfaces, and could be simplified.

The next Chapter contains a description of interfaces that were constructed using `a2b`.

Interface implementations

Chapter 13

During the course of this project, the author has constructed a number of interfaces to perform different types of data transfers. All these interfaces were constructed according to the principles discussed in early Chapters of this thesis, and the more recent of these were constructed using the software tool `a2b`. A selection of these interfaces is described in this Chapter to illustrate these principles, and the use of `a2b` for constructing interfaces from transfer specifications.

In Section 13.1, construction is described of four interfaces which transform two different data sets into a representation suitable for a research database. Another non-communicating interface, described in Section 13.2, transforms data represented according to the Spatial Data Transfer Standard (Geological Survey 1992) into data having a different physical representation defined by the author. This latter interface was constructed by the author to gain experience with the important representation defined by the Spatial Data Transfer Standard. In Section 13.3, construction is described of a pair of communicating interfaces to perform a data transfer similar to the one described in the introduction to Chapter 7.

The next Chapter contains a discussion of future areas for research in light of the author's experience in constructing a number of different interfaces using `a2b`.

13.1 Departmental research database

As part of constructing a database for use in departmental research, the author combined two data sets referred to here as the Dosli data set, and the Census data set. The Dosli data set comprised a subset of values from the Digital Cadastral Database which is being compiled by the Department of Survey, Land and Information (DOSLI). The Dosli data set contains a digital representation of parcels (units of land) and other legal boundaries, and

is approximately 17 Megabytes in size when represented as a text file according to the Gina data file format.

The Census data set, obtained from the New Zealand Department of Statistics, comprised the boundaries of census areas for the Christchurch region. This data set was represented within an Ingres relational database using the three relations `cd`, `polch`, and `chxy`, shown in Figure 13.1. Attribute names forming the primary keys for the relations shown in Figure 13.1 are italicised.

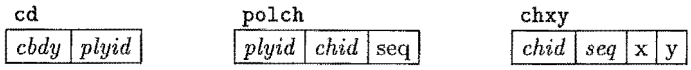


Figure 13.1: Relations constituting the Census relational database

The relation `cd` comprised two attributes: *cbdy*, the official standard unit area code unique to each census area; and *plyid*, a foreign key identifying the polygon used to represent this census boundary. The relation `polch` comprised three attributes: *plyid*, which uniquely identifies each polygon represented in the database; *chid*, a foreign key identifying the chain used to define part or all of the polygon's boundary; and seq, for ordering those chains that defined the polygon's boundary. The third relation `chxy` comprised four attributes: *chid*, which uniquely identifies each chain represented within the database; *seq*, for ordering the points which define a chain; and x and y, which define a point within the chain.

The Dosli and Census data sets were to be combined in such a way that each parcel was associated with the census area within which the parcel was located. Essentially, the goal of the author was to create data for the relations shown in Figure 13.2.

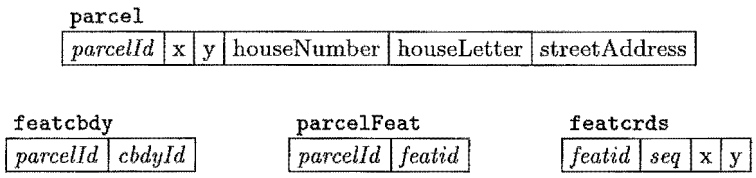


Figure 13.2: Relations for storing the transferred data in the departmental database

The relation `parcel`, comprising the centroid and street address of each parcel in the Dosli data set; `featcbdy` comprising the feature identifier of each parcel, and the standard area unit code of the census area within which this parcel was located; `parcelFeat`, comprising the parcel identifier and the identifier of a feature used to define part of this parcel's boundary; and the relation `featcrds`, comprising the spatial definition of the features which define

parcel boundaries.

The problem of combining these two data sets to produce the three relations `parcel`, `featcbdy` and `featcrds` was divided into four data transfers:

$\mathcal{S}_{Dosli} \xrightarrow{*} \mathcal{D}_{featcrds}$

the transfer of the definition of the parcel boundaries within the Dosli data set $\mathcal{S}_{Dosli}$, into the data values $\mathcal{D}_{featcrds}$ forming the relation `featcrds`;

$\mathcal{S}_{Dosli} \xrightarrow{*} \mathcal{D}_{parcels}$

the transfer of the street address and centroid of the parcels within the Dosli data set, into the data values $\mathcal{D}_{parcels}$ forming the relation `parcels`;

$\mathcal{S}_{census} \xrightarrow{*} \mathcal{T}_{cbdys}$

the transfer of the census data set $\mathcal{S}_{census}$, into the data set $\mathcal{T}_{cbdys}$ comprising the spatial definition and the standard unit area code of each census area; and

$\mathcal{T}_{parcels,cbdys} \xrightarrow{*} \mathcal{D}_{featcbdys}$

the transfer of the data set $\mathcal{T}_{parcels,cbdys}$, comprising the union of the two data sets $\mathcal{D}_{parcels}$ and $\mathcal{T}_{cbdys}$, into the data set $\mathcal{D}_{featcbdys}$, comprising the data values forming the relation `featcbdy`.

In Figure 13.3, an overview is presented of the process by which the two data sets $\mathcal{S}_{Dosli}$ and $\mathcal{S}_{census}$ were transferred to form the three relations `featcrds`, `parcels`, and `featcbdy`, and the interfaces used in this process. These interfaces were generated by `a2b` from the transfer specifications given in Appendix E. Data for the relation `parcelFeat` was taken directly from the text file in which the data set $\mathcal{S}_{Dosli}$ was represented (The `udb-polygon` section of the Gina data file format).

The transfer $\mathcal{S}_{Dosli} \xrightarrow{*} \mathcal{D}_{featcrds}$ comprises the following sequence of transformations:

$$\mathcal{S}_{Dosli} \xrightarrow{decode} \mathcal{T}_0 \xrightarrow{translate} \mathcal{T}_1 \xrightarrow{encode} \mathcal{D}_{featcrds}$$

all of which are performed by the interface `gfc`. Fragments of the data sets $\mathcal{S}_{Dosli}$ and $\mathcal{D}_{featcrds}$ are shown in Figures 13.4 and 13.5, together with representation definitions for these data sets that form a part of the transfer specification from which the interface `gfc` was generated.

The transfer $\mathcal{S}_{Dosli} \xrightarrow{*} \mathcal{D}_{parcels}$ comprises the following sequence of transformations:

$$\mathcal{S}_{Dosli} \xrightarrow{decode} \mathcal{T}_0 \xrightarrow{translate} \mathcal{T}_1 \xrightarrow{encode} \mathcal{D}_{parcels}$$

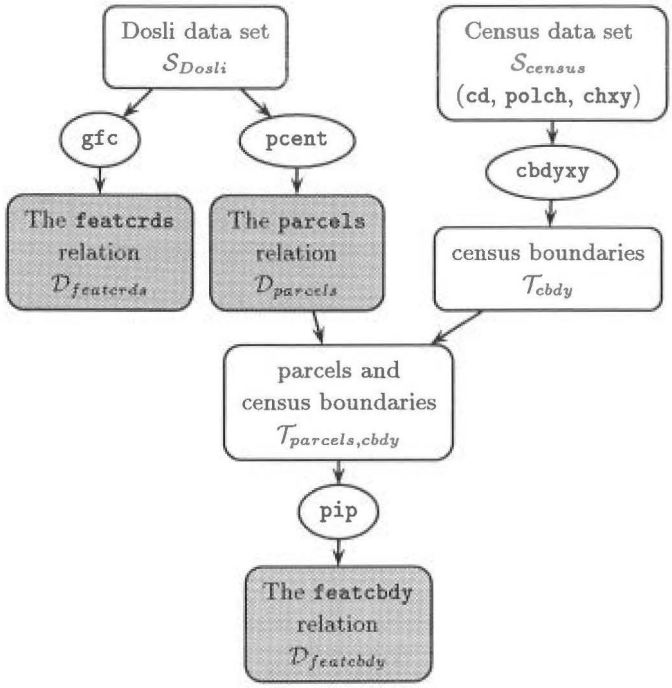


Figure 13.3: Data processing for the research database

all of which are performed by the interface `pcent`. The interface `pcent` extracts the centroid and address of each parcel within the Dosli data set and stores them in another data file. A fragment of the data file produced by `pcent` is shown in Figure 13.6, together with a representation definition for this data set that forms a part of the transfer specification from which the interface `pcent` was generated.

The transfer $\mathcal{S}_{census} \xrightarrow{*} \mathcal{T}_{cbdys}$ comprises the following sequence of transformations:

$$\mathcal{S}_{census} \overset{translate}{\longrightarrow} \mathcal{T}_0 \overset{encode}{\longrightarrow} \mathcal{T}_{polxy} \overset{decode}{\longrightarrow} \mathcal{T}_2 \overset{translate}{\longrightarrow} \mathcal{T}_3 \overset{encode}{\longrightarrow} \mathcal{T}_{cbdys}$$

where

$\mathcal{S}_{census} \overset{translate}{\longrightarrow} \mathcal{T}_0$

is a transformation that results in the three relations `cd`, `polch`, and `chxy` being joined to form a fourth relation `polxy`. The relation `polxy` is created using an SQL `select` statement, which is executed by the Ingres relational database management system;

$\mathcal{T}_0 \overset{encode}{\longrightarrow} \mathcal{T}_{polxy}$

is a transformation of the data contained within the relation `polxy`, into the same data set represented as a text file according to the representation definition shown

```
%dosli
dosli : "udb-feature" '\n' { Feature }* ;
Feature : ("feat" description
        [ ("coor" (point)* '\n')* ]
        [ "text" str '\n' ]
        [ "attr" (( attrValue || ",")+ || ",\natcr")+ '\n' ]);
description: ( id featNonSpatialType layerId networkId featSpatialType
         featGeomSpace gp1 gp2 flow '\n');
attrValue : ([< in | rl | str >]) ;
point : (x y);
featNonSpatialType : << "parcel" | "parcel_bdy" | "r_r_bdy" | "vinculum"
                      | "attr_node" | "r_r_hyd" | "road" | "r_r_arc"
                      | "landmark" | "pte_road" | "pte_rd_bdy"  | "pte_rd_arc"
                      | "frm_road" | "parcel_arc" | "railway" | "parcel_hyd" >>;
featSpatialType: << "l" | "n" | "p" | "pt" >>;
featGeomSpace: < << "xy" | "xyz">> | "xycz" constz > ;
str : string=("\"((\\.)|[^\"\\])*\"" : "\"%s\"");
x,y :  number ;
number : <i | r>;
r, rl, constz,gp1,gp2 : real;
i, in, id,layerId,networkId,flow : integer ;
```

(a) The representation definition for the data set $\mathcal{S}_{Dosli}$

```
udb-feature
feat 709 parcel_bdy 1 1 l xy 0.000000 0.000000 1
coor  2491874.179 5736863.293 2491897.661 5736787.209
feat 710 parcel 1 1 p xy 0.000000 0.000000 1
coor  2481324.166 5736497.367
attr "Lot 7","DP 27852",    0.0738,1,"","","","","","","","","","","","","",
    0.0000,"",
atcr "","",    0.0000,"","","",    0.0000,"","","",    0.0000,"","","",
    0.0000,"4","DALEFIELD DRIVE","17 AUG 1989 00:00:00"
feat 711 r_r_bdy 1 1 l xy 0.000000 0.000000 1
coor  2481344.463 5737573.146 2481349.473 5737591.575
feat 712 parcel_bdy 1 1 l xy 0.000000 0.000000 1
```

(b) A fragment of the data set $\mathcal{S}_{Dosli}$ processed by interfaces `gfc` and `pcent`

Figure 13.4: The data set $\mathcal{S}_{Dosli}$

```
%featcrds
featcrds : (crds)*;
crds:   (id seq x y '\n');
x,y :  real;
seq, id : integer ;
```

(a) The representation definition for the data set $\mathcal{D}_{featcrds}$

```
709 0 2491874.179000 5736863.293000
709 1 2491897.661000 5736787.209000
710 0 2481324.166000 5736497.367000
711 0 2481344.463000 5737573.146000
711 1 2481349.473000 5737591.575000
712 0 2481514.458000 5736391.977000
712 1 2481543.045000 5736403.277000
713 0 2483137.482000 5738409.282000
774 0 2481222.210000 5739176.971000
775 0 2491981.748000 5736501.272000
```

(b) A fragment of the data set $\mathcal{D}_{featcrds}$ produced by the interface gfc

Figure 13.5: The data set $\mathcal{D}_{featcrds}$

```
%parcels
parcels : { Feature }* ;
Feature : ( id ":" x ":" y ":"
            [ houseNumber ] ":" [ houseLetter ] ":" [ streetAddress ]'\n') ;
x,y :  real;
id,houseNumber : integer ;
houseLetter,streetAddress : string=("[A-Z*][^:\n]*" : "%s");
```

(a) The representation definition for the data set $\mathcal{D}_{parcels}$

```
710:2481324.166000:5736497.367000:4::DALEFIELD DRIVE
795:2481449.172000:5737015.221000:43:G:BOWENVALE AVENUE
816:2481146.447000:5739115.764000:111::SOUTHAMPTON STREET
819:2481121.619000:5739141.769000:105::SOUTHAMPTON STREET
837:2491791.119000:5736999.056000:0::TAYLORS MISTAKE ROAD
843:2481368.928000:5737093.041000:27:A:BOWENVALE AVENUE
846:2491743.606000:5736993.985000:61::TAYLORS MISTAKE ROAD
862:2491914.352000:5736623.839000:121::TAYLORS MISTAKE ROAD
865:2481192.657000:5739072.697000:125::SOUTHAMPTON STREET
870:2481267.108000:5739003.961000:149::SOUTHAMPTON STREET
```

(b) A fragment of the data set $\mathcal{D}_{parcels}$ produced by the interface pcent

Figure 13.6: The data set $\mathcal{D}_{parcels}$

```
%polxy
polxy : (idxy)*;
idxy : (cbdy ":" x ":" y);
cbdy : integer;
x,y : real;
```

(a) The representation definition for the data set $\mathcal{T}_{polxy}$

```
8600:2486708.000:5755475.000
8600:2486626.000:5755479.000
8600:2486632.000:5755471.000
8600:2486508.000:5755460.000
8600:2486395.000:5755424.000
8600:2486304.000:5755367.000
8600:2486283.000:5755368.000
8600:2486266.000:5755396.000
8600:2486277.000:5755452.000
8600:2486309.000:5755601.000
```

(b) A fragment of the data set $\mathcal{T}_{polxy}$ processed by the interface cbdyxy

Figure 13.7: The data set $\mathcal{T}_{polxy}$

```
%cbdys
cbdys : (cbdy)*;
cbdy : ("boundary :" id '\n' points '\n');
points : (point)*;
point : ("(" x ", " y ")" '\n');
id : integer;
x,y : real;
```

(a) The representation definition for the data set $\mathcal{T}_{cbdys}$

```
boundary :8600
(2486626.000000, 5755479.000000)
(2486632.000000, 5755471.000000)
(2486508.000000, 5755460.000000)
(2486395.000000, 5755424.000000)
(2486304.000000, 5755367.000000)
(2486283.000000, 5755368.000000)
(2486266.000000, 5755396.000000)
(2486277.000000, 5755452.000000)
(2486309.000000, 5755601.000000)
```

(b) A fragment of the data set $\mathcal{T}_{cbdys}$ produced by the interface cbdyxy

Figure 13.8: The data set $\mathcal{T}_{cbdys}$

```
%featcbdys
featcbdys : (featcbdy)*;
featcbdy : (pid cbdy '\n');
pid,cbdy : integer;
```

(a) The representation definition for the data set $\mathcal{T}_{featcbdys}$

```
710 9110
795 9110
816 9460
819 9460
837 9620
843 9110
```

(b) A fragment of the data set $\mathcal{T}_{featcbdys}$ produced by the interface pip

Figure 13.9: The data set $\mathcal{D}_{featcbdys}$

in Figure 13.7. A fragment of this text file is also shown in Figure 13.7. The data set $\mathcal{T}_{polxy}$ is created using an SQL `copy` statement, which is executed by the Ingres relational database management system;

$\mathcal{T}_{polxy} \stackrel{decode}{\longrightarrow} \mathcal{T}_1 \stackrel{translate}{\longrightarrow} \mathcal{T}_2 \stackrel{encode}{\longrightarrow} \mathcal{T}_{cbdys}$

is a sequence of transformations performed by the interface `cbdyxy`. As part of the transfer, the interface `cbdyxy` removes duplicate points from the spatial definition of the polygons defining the census areas. These duplicate points are the start and end points of consecutive chains that define the boundary of a polygon. A fragment of the data set $\mathcal{T}_{cbdys}$ produced by the interface `cbdyxy` is shown in Figure 13.8, together with a representation definition for this data set that forms a part of the transfer specification from which the interface `cbdyxy` was generated.

The data sets $\mathcal{D}_{parcels}$ and $\mathcal{T}_{cbdys}$ are concatenated using the Unix tool `cat` to form the data set $\mathcal{T}_{parcels,cbdys}$. This data set is transformed into the data set $\mathcal{D}_{featcbdys}$ by the following sequence of transformations:

$$\mathcal{T}_{parcels,cbdys} \stackrel{decode}{\longrightarrow} \mathcal{T}_0 \stackrel{translate}{\longrightarrow} \mathcal{T}_1 \stackrel{encode}{\longrightarrow} \mathcal{D}_{featcbdys}$$

all of which are performed by the interface `pip`. The interface `pip` produces the data set $\mathcal{D}_{featcbdys}$, comprising pairs of values, each pair comprising a parcel identifier and the standard area unit code of the census area within which the parcel lies. Part of the data set $\mathcal{D}_{featcbdys}$ is shown in Figure 13.9, together with a representation definition for this data set that forms a part of the transfer specification from which the interface `pip` was generated.

13.2 Experiences with SDTS

To gain experience with the Spatial Data Transfer Standard (SDTS), discussed earlier in Section 4.4.1, the author transferred a set of data values represented according to the SDTS, into a set of data values having a representation defined by the author. This transfer is described as: $\mathcal{S}_{SDTS} \mapsto \mathcal{D}_{VSDTS}$ where $\mathcal{S}_{SDTS}$ is a set of values conforming to the SDTS and $\mathcal{D}_{VSDTS}$ is a set of values represented in such a way that:

- Explanatory comments are included within the representation. These comments indicate the beginning of the leader fields, directories, and the different records contained within the text file;
- Values of simple data types are represented using however many characters are required by each value, rather than as a number of characters that is fixed for all values of some

type;

- In general, field terminators are replaced by newlines, and unit terminators are replaced by commas.

This representation was designed by the author for the purpose of understanding the ISO 8211 text file representation (ISO8211 1985), and was designed in such a way that only physical data transformations were required to accomplish the transfer $\mathcal{S}_{SDTS} \mapsto \mathcal{D}_{VSDTS}$. Thus, the transfer is described as:

$$\mathcal{S}_{SDTS}(C,\ I,\ P_{ISO8211}) \stackrel{*}{\rightarrow} \mathcal{D}_{VSDTS}(C,\ I,\ P_{VSDTS}).$$

This transfer was accomplished using two interfaces: `fix` and `sdts`. The interface `fix` performed the following sequence of transformations:

$$\mathcal{S}_{SDTS}(C,\ I,\ P_{ISO8211}) \stackrel{decode}{\longrightarrow} \mathcal{T}_0(C,\ I,\ P_0) \stackrel{encode}{\longrightarrow} \mathcal{T}_1(C,\ I,\ P_1)$$

where the data set $\mathcal{S}_{SDTS}$ is transformed into the data set $\mathcal{T}_1(C,\ I,\ P_1)$, which conforms to a physical representation defined by P_1, which differs from the physical representation defined by $P_{ISO8211}$ in that records within the text file representation of the data set $\mathcal{T}_1$ are separated by newline characters. In every other way, the data set $\mathcal{T}_1$ produced by the interface `fix` conforms to the ISO 8211 text file representation used by the SDTS. The interface `sdts` performed the following sequence of transformations:

$$\mathcal{T}_1(C,\ I,\ P_1) \stackrel{decode}{\longrightarrow} \mathcal{T}_2(C,\ I,\ P_2) \stackrel{encode}{\longrightarrow} \mathcal{D}_{VSDTS}(C,\ I,\ P_{VSDTS})$$

where the physical representation of data values $\mathcal{T}_1$ produced by the interface `fix` is transformed into the data set $\mathcal{D}_{VSDTS}$.

The transformation performed by the interface `fix` was necessary because of the method defined by the ISO 8211 standard for representing data values within a text file. This method appears to have been designed to facilitate the processing of these text file representations by programs that decode and encode data values within variable length records and fields, where the length of these records and fields is given within the file.

This approach to representing data values within a text file cannot be elegantly specified by a *single* representation definition which is expressed using the notation described in Section 7.2. Modifying the method of representation so that newline characters are used to terminate particular types of data values does allow a reasonable representation definition to be specified. The purpose of the interface `fix` is therefore to transform a data set conforming

to the SDTS into a data set conforming to this slightly different representation.

```
#include <stdio.h>

main(argc,argv)
     int argc; char **argv;
{
      char *b, sz[5];
      int n,bsz,atoi();
      FILE *ifd, *ofd;

      ifd = fopen(argv[1],"r");
      ofd = fopen(argv[2],"w");
      while((n = fread(sz,5,1,ifd)) > 0)
      {
            bsz = atoi(sz) - 5;
            b = (char *)malloc(bsz);
            fread(b,bsz,1,ifd);
            fwrite(sz,5,1,ofd);
            fwrite(b,bsz,1,ofd);
            fwrite("\n",1,1,ofd);
            free(b);
      }
}
```

(a) The interface `fix`

```
%a2b sdts to t1

%sdts
sdts : ((#recordLength record))*;
recordLength : integer:5;
record : string:recordLength - 5;

%t1
t1 : ((#recordLength record) "\n")*;
recordLength : integer:5;
record : string:recordLength - 5;

%sdts2t1 %c{
sdts2t1(s,dp)
    struct type_SDTS_Sdts *s;
    struct type_T1_T1 **dp;
{
    *dp = (struct type_T1_T1 *) s;
}
%}
```

(b) Proposed transfer specification for the interface `fix`

Figure 13.10: Construction of the interface `fix`

The `fix` interface was constructed by hand, and is shown in Figure 13.10(a). Figure 13.10(b) shows a transfer specification from which an equivalent interface ought to be generated. The present version of `a2b` does not allow integer expressions that specify a fixed width text file representation for simple types. That is, the type definition `record : string:recordLength - 5;` is illegal. Consequently, the interface `fix` cannot be generated from this transfer specification.

The `fix` interface produces text files such as the one shown in Figure 13.11. The dots '.....' within this Figure indicate where parts of this text file have been omitted. The SDTS data files processed by the `fix` interface were supplied by the U.S. Geological Survey, together with the report on the SDTS (Geological Survey 1992).

The data set $\mathcal{T}_1$ shown in Figure 13.11 was processed by the `sdts` interface to produce the data set $\mathcal{D}_{VSDTS}$, shown in Figure 13.12. Fragments of the transfer specification from which the interface `sdts` was generated are presented in Figure 13.13, with the complete specification being given in Appendix E.1. The dots '...' in Figure 13.13 indicates where

```
003412L    0600080    3404000002600000010300026CATD0690056CATX0660125CATS0700191;
0000;&CHESSY TEST SET WVS;0100;&DDF RECORD IDENTIFIER&&;2600;&CATALOG/DIRECTORY&
*MODN!NAME!TYPE!VOLM!FILE!RECD!COMT&(5A,I,A).....................
01654 D      00267    340400010020000CATD0660002CATD0720068.................1;WVS
 CATD&WVS CATD&Catalog/Directory&CHESSY-WVS&CAT.DDF&1&10/31/88;WVS CATD&WVS CATX
&Catalog/Cross-Reference&CHESSY-WVS&CAT.DDF&2&10/31/88;
00233 D      00058    340400010020000CATS0510002CATS1220053;3;WVS CATS&WVS CATD&Ca
talog/Directory&*&*&*&10/31/88;WVS CATS&FILE&Composite&Chesapeake Bay Area&NJ 18
-8,11 Joint Operations Graphics (JOG) 1:250000 scale&Shoreline&10/31/88 ;
```

Figure 13.11: Fragments of the data set $\mathcal{T}_1$ produced by the interface `fix`

parts of the specification have been omitted.

In defining the specifications for the transfer $\mathcal{S}_{SDTS} \xrightarrow{*} \mathcal{D}_{VSDTS}$, the author learned a great deal about the method of data representation defined by the ISO 8211 standard and the SDTS, and the capabilities of `a2b`. The ISO 8211 standard defines a comprehensive method of representing data values within a text file. Not only can the values themselves be represented within this text file, but descriptions of these values are also contained within the text file.

Such a comprehensive representation complicates interface construction because the exact type and representation of data values can be determined only after reading the text file. Thus, either the decoder module within an interface must be capable of using the data definitions given within the text file to interpret the data values, also given within this text file, or the decoder module must be constructed after reading the data definitions given within the text file.

The only information given within a text file that can be interpreted by a decoder module generated by `a2b` are integer values that determine either the number of values in an N-set or an N-sequence (Section 7.2.1.2), or the number of characters that are used to represent a data value (Section 7.2.2.1). Therefore, any additional information within the text file that describes the types of data values represented in the same text file must be interpreted by a translator module.

The ISO 8211 standard is also complex because the representation of data values may depend on a fixed number of characters, the use of unit and field terminators to delimit the representation of data values, or some combination of both techniques. In some cases, the use of N-sets, N-sequences, and simple types represented using some fixed number of characters within a representation definition is insufficient to allow these text files to be decoded. Hence, interfaces such as `fix`, as described above, have to be constructed.

```
DDR Leader:
341 2"L"" "" "" "6 80 3 4"0"4
DDR Directory:
"0000"26 0
"0001"30 26
"CATD"69 56
"CATX"66 125
"CATS"70 191
DDR Data descriptive area:
"0000;&"
name:"CHESSY TEST SET WVS"
"0100;&"
name:"DDF RECORD IDENTIFIER"
"2600;&"
name:"CATALOG/DIRECTORY"
Label:"*MODN!NAME!TYPE!VOLM!FILE!RECD!COMT"
Fctrl:"(5A,I,A)"
.............
.............
DR Leader:
1654" ""D"267 3 4"0"4
DR Directory:
"0001"2 0
"CATD"66 2
"CATD"72 68
.............
.............
DR Data descriptive area:
"1"
"WVS CATD","WVS CATD","Catalog/Directory","CHESSY-WVS","CAT.DDF","1","10/31/88"
"WVS CATD","WVS CATX","Catalog/Cross-Reference","CHESSY-WVS","CAT.DDF","2","10/31/88"
.............
.............
DR Leader:
233" ""D"58 3 4"0"4
DR Directory:
"0001"2 0
"CATS"51 2
"CATS"122 53
DR Data descriptive area:
"3"
"WVS CATS","WVS CATD","Catalog/Directory","*","*","*","10/31/88"
"WVS CATS", "FILE", "Composite", "Chesapeake Bay Area",
  "NJ 18-8,11 Joint Operations Graphics (JOG) 1:250000 scale","Shoreline","10/31/88 "
```

Figure 13.12: Parts of the data set $\mathcal{D}_{VSDTS}$ produced by the sdts interface

```
%a2b sdts to tmp

%sdts
sdts  : (ddr (dr)* );
ddr : (ddrLeader ddrDirectory "\036" (field1 "\036")* "\n");
dr : (drLeader drDirectory "\036" (field2 "\036")* "\n");
ddrLeader : (   recordLength interchangeLevel leaderIdentifier
                ... #szFieldLength #szFieldPosition reserved2 #szFieldTag);
ddrDirectory: (directoryEntry)^(addrsdda - 24) /
                              (szFieldLength + szFieldPosition + szFieldTag);
directoryEntry: ( fieldTag fieldLength fieldPosition);
drLeader : (    drLength reserved1 drlIdentifier [reserved3] #baseAddress
                [reserved4] #szFieldLength #szFieldPosition reserved2,
                #szFieldTag );
drDirectory: (directoryEntry)^(baseAddress - 24)
                        / (szFieldLength + szFieldPosition + szFieldTag);
field1 : < (fileControlField name  [ "\037" tagPairs ])
         | (fieldStruc [name] "\037" [label] "\037" [fctrl]) >;
field2 : (unit || "\037")*;
unit : ([data]);
fieldStruc : < elementaryField | bitField | collectionField > ;
fileControlField:               string = ("0000;&":"%s");
elementaryField:                string = ("0[1-4]00;&":"%s");
bitField:                       string = ("0500;&":"%s");
collectionField:                string = ("[1-2][0-6]00;&":"%s");
tagPairs,name,label,fctrl,data,
data1,data2:                    string = ("[^\037\036\;\&\n]+":"%s");
recordLength, addrsdda,baseAddress,drLength:            integer:5;
interchangeLevel,szFieldPosition,szFieldLength,szFieldTag: integer:1;
leaderIdentifier,reserved1,reserved2,drlIdentifier,
applicationIndicator,extensionIndicator:                string:1;
reserved4,extdCharSetIndicator:                         string:3;
fieldControlLength:                                     integer:2;
fieldLength:                                            integer:szFieldLength;
fieldPosition:                                          integer:szFieldPosition;
fieldTag:                                               string:szFieldTag;
reserved3:                                              string:5;

%tmp
tmp : (ddr (dr)* );
...

%sdts2tmp %c{
sdts2tmp(s,dp) struct type_SDTS_Sdts *s; struct type_TMP_Tmp **dp;
{
    *dp = (struct type_TMP_Tmp *) s;
}
%}
```

Figure 13.13: Fragments of the transfer specification for the interface sdts

Specifying a representation definition for data conforming to the SDTS was difficult. However, the author found a2b could be used to generate interfaces for processing data files conforming to the SDTS, which defines a complex data representation.

13.3 Construction of communicating interfaces

To illustrate the use of a2b for constructing a pair of communicating interfaces, consider the following transfer:

$$\mathcal{S} \xrightarrow{*} \mathcal{N}_S \overset{communicate}{\rightsquigarrow} \mathcal{N}_D \xrightarrow{*} \mathcal{D}$$

where the data sets $\mathcal{S}$ and $\mathcal{D}$ are the same as those described in the introduction of Chapter 7, but are located on different computer systems, and the data sets $\mathcal{N}_S$ and $\mathcal{N}_D$ conform to the same conceptual and implementation schemas as the data set $\mathcal{D}$ but conform to a different physical schema corresponding to the network representation used by ISODE.

The specification for the above transfer is shown in Figure 13.14, and was processed by a2b to generate the required pair of communicating interfaces. Note that the %netRep representation definition differs from the destination representation definition only in that there are no string literals defined in the %netRep representation definition. These literals are not required as values conforming to the %netRep representation definition are not going to be represented within a text file.

13.4 Conclusions

Construction of the interfaces discussed in this Chapter, and others not described, has illustrated the usefulness of the underlying principles presented in this thesis. In Chapter 3, a notation was defined for describing a transfer as a sequence of transformations, each transforming data to either conform to a different conceptual, implementation, or physical schema, or to be located on a different computer system. Use of this notation has greatly assisted discussion of the different elements of the transfer process, and definition of specific transfers such as those described in this Chapter.

Using this notation, the overall structure of any transfer can be defined as a combination of transformations, each being one of four types: encode, decode, translate, or communicate. Given this definition, the form and complexity of the interfaces required to perform this transfer can be determined. Consider, for example, the transfer $\mathcal{S}_{census} \xrightarrow{*} \mathcal{T}_{cbdys}$ discussed in Section 13.1.

```
%a2b source via netRep to destination

%source
source:          (feature || "\n" )* ;
feature:          ("feat" (id type) "\ncoor" (point)*);
point:                (x y);
id, x, y:         integer;
type:                << "L" | "P" >>;

%netRep
netRep:          ( < (( id #nPts minBoundRect) (point)^nPts) | ( id x y ) > )*;
point:                (x y);
minBoundRect:         (mnx mny mxx mxy);
id, nPts, x, y, mnx, mny, mxx, mxy: integer;

%destination
destination:          (
                    < "Line " (( id #nPts minBoundRect) "\n" (point || "\n")^nPts)
                    | "Point " ( id x y )> "\n")*;
point:                (x y);
minBoundRect:         (mnx mny mxx mxy);
id, nPts, x, y, mnx, mny, mxx, mxy: integer;

%source2netRep %mira{
source2netRep data
  = map src2nrObj data
    where
      src2nrObj ((id,0),pts)
        = NetRep_choice_1 ((id, # pts, mbr), pts)
          where
            thePts = foldl dopts ((10000, 10000, -10000, -10000), []) pts
            mbr = extract_mbr thePts where extract_mbr (a,b) = a
      src2nrObj ((id,1),(x,y):pts)
        = NetRep_choice_2 (id, x, y)
      dopts ((mnx, mny, mxe, mxn), xys) (x,y)
          = ((min2 x mnx, min2 y mny,
              max2 x mxe, max2 y mxn), (x,y):xys)
%}

%netRep2destination %c{
netRep2destination(nr, dp)
  struct type_NETREP_NetRep *nr;
  struct type_DESTINATION_Destination  **dp;
{
  *dp = (struct type_DESTINATION_Destination *) nr;
}
%}
```

Figure 13.14: Transfer specification for a pair of communicating interfaces

This transfer is described in full as:

$$\mathcal{S}_{census} \overset{translate}{\longrightarrow} \mathcal{T}_0 \overset{encode}{\longrightarrow} \mathcal{T}_{polxy} \overset{decode}{\longrightarrow} \mathcal{T}_2 \overset{translate}{\longrightarrow} \mathcal{T}_3 \overset{encode}{\longrightarrow} \mathcal{T}_{cbdys}$$

and requires two interfaces: the first, a sequence of SQL statements, to perform the transformations $\mathcal{S}_{census} \overset{translate}{\longrightarrow} \mathcal{T}_0 \overset{encode}{\longrightarrow} \mathcal{T}_{polxy}$. The second, generated from a transfer specification using a2b, to perform the transformations $\mathcal{T}_{polxy} \overset{decode}{\longrightarrow} \mathcal{T}_2 \overset{translate}{\longrightarrow} \mathcal{T}_3 \overset{encode}{\longrightarrow} \mathcal{T}_{cbdys}$.

In comparison, the transfer $\mathcal{S}_{Dosli} \overset{*}{\rightarrow} \mathcal{D}_{parcels}$, which can be described in full as:

$$\mathcal{S}_{Dosli} \overset{decode}{\longrightarrow} \mathcal{T}_0 \overset{translate}{\longrightarrow} \mathcal{T}_1 \overset{encode}{\longrightarrow} \mathcal{D}_{parcels}$$

is performed by one interface, generated from a transfer specification using the software tool a2b, and comprises one translation, instead of the two required by the transfer $\mathcal{S}_{census} \overset{*}{\rightarrow} \mathcal{T}_{cbdys}$. In general, transfers involving data translations or movements of data are more complex than those without these types of transformations.

Dividing a transfer into a sequence of transformations has led to a formal definition of those transfers that are also divisible into smaller definitions. These smaller definitions, or a slightly modified form of these definitions, can be used to define potentially many different transfers. Reuse of these definitions (with or without modification) reduces the time taken to construct interfaces, which has been a goal of this thesis.

Reuse of parts of a transfer specification was illustrated in Section 13.1, where the representation definition for the data set $\mathcal{S}_{Dosli}$, shown earlier in Figure 13.4, formed the basis for part of the formal definition of two transfers: $\mathcal{S}_{Dosli} \overset{*}{\rightarrow} \mathcal{D}_{featcrds}$, and $\mathcal{S}_{Dosli} \overset{*}{\rightarrow} \mathcal{D}_{parcels}$. Similarly, the representation definitions called `source` and `destination`, shown earlier in Figure 7.2, were used to define parts of two transfers. One, $\mathcal{S} \overset{*}{\rightarrow} \mathcal{D}$, which required no movement of data through a communications network, was discussed throughout Chapter 7. The other, $\mathcal{S} \overset{*}{\rightarrow} \mathcal{N}_S \rightsquigarrow \mathcal{N}_D \overset{*}{\rightarrow} \mathcal{D}$, which required data to be moved through a communications network, was discussed in Section 13.3.

Future research and development

Chapter 14

Implementation and use of the software tool a2b has led the author to consider further developments to a2b, and areas for future research that is expected to result in a more powerful software tool. Future developments to be discussed in Section 14.1 include extending the notation defined earlier in Section 7.2 by defining a new simple type, and generally improving the output generated by a2b. Defining a variety of interface templates is discussed in Section 14.2 and increasing the assistance for debugging the generated interfaces is discussed in Section 14.3.

Section 14.4 is a discussion of the research and development of a translation environment for the existing tool a2b, and concludes by briefly discussing a more ambitious and long term goal of developing a more powerful software tool which generates translator modules from a comparison of the representation definitions to which the data conforms before and after the translation. The Chapter is concluded in Section 14.5 with a discussion of using a2b for generating interfaces which transfer a variety of data.

14.1 Future developments

The point has been reached where the author's experience in constructing and using a2b suggests ways in which this tool can be improved. Although some of these improvements appear to be straight-forward, in practice performing these changes are likely to be an awkward and time-consuming task.

The software tool a2b was developed by the author as a research tool. Consequently, interest lay in the functionality of this tool rather than in providing extensive error checking.

Error checking when processing a transfer specification should now become an important part of this processing. Examples of the error checking that a2b should perform include:

- checking that all representation definitions and implied translation definitions in an interface definition are specified within a transfer specification; and
- checking that all user-defined types referenced within a representation definition are actually defined within this definition.

The notation for defining representations described earlier in Section 7.2 needs to be extended to include a new simple type, provisionally called `bytes`. Unlike the representation of values of the simple type `string`, which is based on the use of delimiters such as quotes, the representation of values of the type `bytes` would not use delimiters. Appendix D contains an explanation of other necessary improvements to the implementation of a2b which are associated with the notation processed by this tool.

Other modifications of a2b can be seen that should improve the interfaces generated by this tool. For example, the ASN.1 abstract syntaxes generated by a2b do not make use of the keyword `implicit`. Use of this keyword in the definition of data types would result in a more compact physical representation of these types of values. That is, fewer bytes would be used to represent a value. This more compact representation could be important as interfaces can be used to transfer very large volumes of data.

14.2 Interface templates

Use of templates as the basis for the interfaces generated by a2b offers the potential for constructing a wide range of interfaces without modifying a2b. Different interface templates could be defined to perform a variety of tasks over and above the data transfers themselves. For example, experience with a primitive data directory service, described earlier in Section 12.2, suggested the idea of developing a much more sophisticated and useful directory service using `quipu`, an implementation of the X.500 directory service provided by ISODE. Use of X.500 could be incorporated within the communicating interfaces generated by a2b by modifying the templates for communicating interfaces to include functions for searching and displaying entries in the X.500 directory.

To fully exploit the use of different templates, a change is required in the method used by a2b for determining where in a template to insert the various parts of an interface generated by a2b. The author suggests adapting the technique used by the software tool RCS (Revision Control System) (Tichy 1985) for maintaining version numbers and revision dates within any source code managed by this tool. RCS replaces keywords within the source code with

information corresponding to these keywords. For example, the keyword `$Id$` would be replaced with a marker containing the filename, revision number, date, time, author, and state.

A set of keywords may be defined for the parts of an interface generated by `a2b` and positioned in appropriate places throughout the various interface templates. Examples of such keywords are: `$Include files$`, to be replaced with the statements for including the various header files generated by `a2b`; `$Variable declarations$`, to be replaced with the declaration of variables used for memory-resident representations of the transferred data; and `$Main interface routine$`, to be replaced with the calls to the various modules that transform the data.

14.3 Assistance for debugging

The author's experience has shown that definition of transfer specifications is an iterative process. An initial transfer specification often has to be modified after the interface generated from this specification fails to transfer data successfully. Any interface should therefore provide information describing why the transfer was unsuccessful. Often, for example, there are difficulties in decoding the data set from the source text file representation. Information provided by the interface explaining which part of the text file could not be decoded would assist in modifying the transfer representation from which the interface is generated.

Interfaces constructed using the author's present methods produce some debugging information using the facilities supplied either by software tools that are used by `a2b`, such as `bison` and `flex`, or by error handling routines inserted into the interface by `a2b`. The level of debugging provided by an interface is given as a flag to `a2b` at the time of generating the interface. These flags indicate which of four levels of debugging assistance is to be provided:

scanner

all debugging facilities provided by `flex` are included within the scanner generated from each representation definition within a transfer specification;

parser

all debugging facilities provided by `bison` are included within the parser generated from each representation definition within a transfer specification;

decoder

all debugging facilities for both the scanner and parser are included for every decoder module within the generated interface;

responder

all debugging facilities provided by a2b are inserted into the responder. These facilities were described in Section 11.2.3.2.

Interfaces generated by a2b are always compiled to allow source code debugging of these interfaces using debuggers such as gdb (Stallman & Pesch 1989) and dbx (Sun 1988).

The debugging assistance provided by a generated interface could be improved in two ways. First, allow the level of debugging to be specified for each representation, perhaps within the options section of a representation definition. For example:

```
%a2b src to dest
%src
@debug scanner
src : ... ;
%dest
dest : ... ;
```

Second, provide a flag '-g', used when invoking a2b, to indicate that the interface is to be compiled for source level debugging. More sophisticated debugging techniques are expected to be necessary as part of developing an a2b translation environment, and for diagnosing errors that occur while using communicating interfaces.

14.4 The a2b translation environment

A long term goal of the author is to develop a translation environment that simplifies the construction of efficient translator modules. This goal will be pursued in three ways: by developing a methodology which simplifies the definition of data translations; by devising a method of converting a translation definition into functions implemented using a programming language such as C; and by augmenting the functionality of other modules, such as encoders and decoders, to reduce the work required to translate the data.

14.4.1 Simplifying the definition of data translations

Definition of data translations may be simplified by adapting techniques being developed for solving the problem of schema integration. Schema integration is 'the activity of integrating the schemas of existing or proposed databases into a global, unified schema' (Batini, Lenzerini & Navathe 1986). For example, one important step in both schema integration and data translation is to compare the different schemas to determine any equivalent representations for data values. Techniques used in the process of schema integration for comparing

different schemas, may be adapted and used in defining the translation of data values from one representation to another.

When describing the application of schema integration techniques for merging data sets from two different geographical information systems, Nyerges (1989) also commented on the possible use of these techniques for translating data during a transfer. Nyerges noted that in schema integration the problem is to define one concise schema that specifies a representation for all the information contained within the data sets to be integrated. In contrast, there is little if any opportunity to modify the schemas defining the representation of data before and after a transfer. Consequently, as described earlier in Section 3.2.1.1, there is the chance that information will be lost during a transfer because of incompatibilities among the schemas that define the representation of data before and after transfer.

As well as simplifying data translation by adapting techniques developed for schema integration, future development of the `a2b` translation environment should include design of a data model that provides the structures for representing the data to be transformed, and a collection of operations to transform data in this representation. Use of the relational model for this purpose was discussed in Sections 5.2.3 and 5.2.4. Given the use of sets and sequences described earlier in Section 7.2 for defining data representations, the author would like to examine the possibilities of developing a data model based on sets and sequences, and operators for transforming sets and sequences of data values.

Operators for transforming sets should be based on the relational operators such as project, select, and join. Operators for transforming sequences could be similar to those operators provided by the Miranda programming language for manipulating lists. Examples are `hd` for selecting the first element of a list, `tl` for selecting all but the first element of the list, and `filter` which is applied to one list to create another comprising elements of the first that satisfy some criteria. The usefulness of these, and other operators provided by the Miranda language and the concise translations that may be specified using this language, would encourage the author to develop a notation for the `a2b` translation environment which is similar to Miranda.

Ultimately, the author would like to remove the `a2b` translation environment and extend the notation described earlier in Section 7.2 for defining data representations. The idea is to process transfer specifications expressed in this extended notation using a new and more more powerful software tool that should be able to automatically construct translator modules. The sequence of operations necessary to transform data values between two representations is constructed from a comparison of the definition of these representations made by this software tool. Although this is an ambitious goal, the author believes techniques being developed in areas such as artificial intelligence, expert systems, and schema integration, will make such

a goal achievable in the long-term.

14.4.2 Constructing efficient translator modules

If interfaces are to frequently move data through a communications network, then efficient translator modules will be needed. In the future, constructing efficient translator modules will be accomplished by a component of a2b that generates a translation function, expressed using a programming language such as C, from a translation definition.

The translation function generated from a translation definition would comprise calls to library functions which perform the operations used in the translation definition. The translation would be defined in terms of the data types constituting the representation definitions to which the data conforms before and after translation. References in the translation definition to data types defined within these representations would be converted into references to the corresponding C data structures for representing these types of values.

The time taken to translate data during a transfer is expected to be reduced if parts of a translation can be performed by modules other than the translator. For example, suppose that the cardinality of a set is needed during a data translation, and that members of this set are to be read by a decoder, the decoder could count the number of members while creating the set. The cardinality of sets and sequences, and other information such as the minimum bounding rectangles for spatial objects forms can be collected by the decoder and passed on to the translator.

Improving the efficiency of the generated translator modules will complicate the construction of the software tool which generates these translator modules. Similarly, improving the code generated by a compiler is achieved by increasing the complexity of any optimisation performed by this compiler. Since optimisation by a compiler can be 'complex, expensive, and sometimes unsafe' (Fischer & LeBlanc Jr. 1988), the author believes that the same would be true for many of the improvements made to a translator generated by a software tool such as a2b. Therefore, a compromise would have to be reached between the complexity of the software tool generating the translator modules, and the efficiency of the translator modules generated by this tool.

14.5 The general data transfer problem

Although the author developed a2b specifically for constructing interfaces that transfer geographical data, a2b can be used to construct interfaces that will transfer *any* type of data set between two different text file representations. For example, a2b was used by the author

to constructed an interface for transferring an index describing LaTeX style files into a representation that allowed this index to be used in a relational database. This data transfer is described in full as:

$$\mathcal{S}(C_S,\ I_S,\ P_S) \stackrel{decode}{\longrightarrow} \mathcal{T}_0(C_S,\ I_S,\ P_0) \stackrel{translate}{\longrightarrow} \mathcal{T}_1(C_D,\ I_D,\ P_1) \stackrel{encode}{\longrightarrow} \mathcal{D}(C_D,\ I_D,\ P_D)$$

where $\mathcal{S}$ is the index in its original text file representation, as shown in Figure 14.1(a), and $\mathcal{D}$ is the index represented within an SQL script for loading into a relational database, as shown in Figure 14.1(b).

The interface to perform this transfer was generated from the transfer specification given in Figure 14.2. The dots '...' in this Figure indicates where parts of this specification have been omitted.

Experience in constructing an interface to perform this transfer shows that `a2b` can be applied to constructing interfaces that transfer data sets other than those originating from geographical information systems. For example, interfaces may be constructed that transfer databases or electronic manuscripts between different text file representations. Approaches by other researchers to construction of interfaces for these types of transfers were discussed in Chapter 5.

Use of `a2b` to construct interfaces that transfer a wide range of data may be influenced by the future development of the `a2b` translation environment, described earlier in Section 14.4. This environment may be developed either to provide a high level of assistance for translating data sets of a particular type, such as geographical data sets, or to provide a lower level of assistance for transforming a much wider range of data sets that includes geographical data sets, electronic manuscripts and so on.

Overall, the existing implementation of `a2b` has demonstrated that the approach to interface construction presented in this thesis is viable, and has provided valuable experience and insights that can be used to improve future versions of this and other more powerful software tools.

```
Name: amstex.tex
Description: AMS-TeX is the American Mathematical Society's macros for simplify
ing the typesetting of complex mathematical material. AMS-TeX is meant to be us
ed with plain TeX.  For corresponding support for LaTeX, see AMS-LaTeX.
Keywords: AMS-TeX, math
Author: American Mathematical Society <Tech-Support@math.ams.org>
Supported: yes
Latest Version: v2.1, 5 Apr 1991
Archives: e-math*, ymir, utrecht, aston, stuttgart
Note: The AMS-TeX distribution includes a readme file, a user's guide (amsguide
.tex) and installation notes (amstinst.tex).
See also: AMS-LaTeX

...
```

(a) Part of the index before transfer ($\mathcal{S}$)

```
create table texindx(
        name varchar(40),
        description varchar(500),
        keywords varchar(100),
        archives varchar(100),
        author varchar(100),
        latestVersion varchar(100),
        supported char(4),
        seeAlso varchar(100),
        note varchar(500)
)\p\g
insert into texIndx values(
        " amstex.tex",
        " AMS-TeX is the American Mathematical Society's macros for simplify
ing the typesetting of complex mathematical material. AMS-TeX is meant to be
 used with plain TeX.  For corresponding support for LaTeX, see AMS-LaTeX.",
        " AMS-TeX, math",
        " e-math*, ymir, utrecht, aston, stuttgart",
        " American Mathematical Society <Tech-Support@math.ams.org>",
        " v2.1, 5 Apr 1991",
        " yes",
        " AMS-LaTeX",
        " The AMS-TeX distribution includes a readme file, a user's guide (am
sguide.tex) and installation notes (amstinst.tex).")\g
...
\q
```

(b) Part of the index after transfer ($\mathcal{D}$)

Figure 14.1: Fragments of the style index before and after transfer

```
%a2b texx to texdb

%texx
texx : { entry }* ;
entry: ( "Name" name (attribute)* );
attribute : < "Description" description | "Keywords" keywords |
              "Archives" archives | "Author" author |
              "Latest Version" latestVersion | "Supported" supported |
              "See also" seeAlso | "Note" note > ;
name,description,keywords,archives,author,
latestVersion,supported,seeAlso,note  : string = (":.*\n" : ":%s\n");

%texdb
texdb :         "create table texindx(\n"
                "\tname varchar(40),\n\tdescription varchar(500),\n"
                "\tkeywords varchar(100),\n\tarchives varchar(100),\n"
                "\tauthor varchar(100),\n\tlatestVersion varchar(100),\n"
                "\tsupported char(4),\n\tseeAlso varchar(100),\n"
                "\tnote varchar(500)\n)\\p\\g\n"
        { entry }* "\\q" ;
entry:          "insert into texIndx values(\n"
       ( "\t" name ",\n\t" description ",\n\t" keywords
         ",\n\t" archives ",\n\t" author ",\n\t" latestVersion
         ",\n\t" supported ",\n\t" seeAlso ",\n\t" note ) ")\\g\n" ;
name,description,keywords,archives,author,latestVersion,...   : string;

%texx2texdb %c{
#include "define.h"
texx2texdb(s, dp)  struct type_TEXX_Texx *s; struct type_TEXDB_Texdb **dp;
{
    struct type_TEXDB_Texdb *d = NULL;
    struct type_TEXX_CasSO *a;
    struct type_TEXDB_Entry *da;

    for (; s; s = s->next) {
          nextValue(d,dp,type_TEXDB_Texdb);
           ...
          for (a = s->member_TEXX_0->casSO; a; a = a->next)
                switch (a->element_TEXX_0->offset) {
                      case type_TEXX_Attribute_description : {
                            da->description = a->element_TEXX_0->un.description;
                            break;
                      }
                      ...
                }
    }
}
%}
```

Figure 14.2: Transfer specification for transferring the LaTeX style index

Conclusions

Chapter 15

This thesis is concerned with the problem of constructing interfaces that transfer geographical data from one geographical information system to another. The ability to transfer, and therefore to share, geographical data is important because many organisations have a common use for large amounts of geographical data. The definitions of coastal, national, regional and property boundaries are, for example, used by utility organisations as a framework for their power, drainage and telephone networks, and by local and national government organisations for administration. Given the considerable overlap of usage, transferring the same data set to many different organisations avoids duplicating the expensive task of data capture.

The functionality and structure of interfaces have changed because interfaces are beginning to use communications networks for moving geographical data sets from one computer system to another. Compared with sending geographical data through the post on magnetic tape, use of communications networks by interfaces can greatly increase the accessibility of data, and the speed with which this data can be transferred. The added functions and more complex structure of these interfaces increases the need for methods that will simplify the construction of interfaces.

Important contributions to simplifying the construction of interfaces made in this thesis include: a better understanding of the methods used to represent geographical data and of the data transfer process; the definition of notations for concisely describing data transfers in terms of the data transformations that are required; and design and development of a software tool for generating interfaces from a formal definition of the data transfer that is to be performed.

A data set conforms to a representation definition and is located on some computer system. Representation of geographical and other types of data may be considered at different levels of abstraction. Three levels of abstraction, similar to those described by Elmasri &

Navathe (1989), were defined in this thesis:

the conceptual level

at which the relevant properties and relationships of the real world phenomena are defined, without considering the method of storing these properties and relationships in a computer;

the implementation level

at which the properties and relationships defined at the conceptual level are expressed using the data types and operations defined by one or more implementation data models. Implementation data models form the basis for constructing geographical information systems; and

the physical level

at which the data types defined by an implementation data model are expressed using the definition of data structures and algorithms provided by physical data models for representing and manipulating data values. Physical data models comprise data structures and operations which are typically defined using pseudo-code.

In the author's experience, few data representations to which data conforms before and after transfer are formally defined. Those formal definitions that are available often do not include conceptual schemas. Without complete definitions of the data representations, determining the data transformations needed to transfer data between these representations is often a matter of trial and error.

The author has concluded that providing a notation similar to A2B, which is defined in this thesis for specifying data transfers, is an important step in developing formal definitions of data representations. The notation A2B allows a concise machine-readable definition of both the physical and implementation schemas of a representation definition. Extension of this notation for defining aspects related to the conceptual schema may allow these definitions to be compared, perhaps in the future automatically by software. This comparison would be used by software that generate interfaces to determine the data transformations necessary to transfer data from one representation to another.

The author has specified a data transfer as one or more data transformations, each either modifying data values to conform to a different representation definition, or moving data values to a different computer system. Accordingly, an interface is structured into modules, each performing a data transformation which forms part of the data transfer.

A data transfer is formally defined by a *transfer specification* comprising: definitions of the representations used for the data before, during, and after the transfer; and definitions

of the transformations required to either modify the data to and from these representations, or to move data from one computer system to another computer system. The approach to interface construction suggested in this thesis has been to generate interfaces from these transfer specifications.

To simplify the definition of a transfer, and thereby simplify the construction of interfaces, a specification is structured to allow parts of existing transfer specifications to be used as the basis for specifying new transfer specifications. For example, the definition of the Gina text file representation contained within an existing specification for the transfer of data from the Gina representation to the Spatial Data Transfer Standard (SDTS) representation can be used as a part of the new specification for a transfer of data from the DLG representation to the Gina representation.

The author found that a definition of a text file representation could often be used in many transfer specifications. Consequently, the time taken to create these transfer specifications and generate the necessary interfaces was greatly reduced. The author also found existing representation definitions easy to modify. Therefore, a general definition of a representation could be quickly tailored for a particular data set. For example, within the Gina text file representation there is a field for representing one of many feature codes. The available feature codes are specific to a data set and could easily be changed for transferring different data sets.

A new software tool called `a2b` was designed and constructed by the author to generate interfaces from transfer specifications. The software tool `a2b` was developed in accordance with the following principles:

1. To use existing software tools wherever possible to construct parts of an interface;

2. To use communication protocols defined by international standards for governing the movement of data from one computer system to another through a communications network; and

3. To use templates that define those aspects which are common to all interfaces, and which can be easily modified to allow interfaces to be generated which perform other tasks in addition to data transfers.

The software tools `bison`, `flex`, `rosy`, and `pepsy` are used by `a2b` to construct parts of the interface. The run-time facilities provided by the ISO Development Environment (ISODE) were used in interfaces generated by `a2b` that move data through a communications network. Use of existing software tools avoided 'reinventing the wheel' and allowed more time to be spent on developing those parts of `a2b` which were specific to interface construction.

Coordinating the use of existing software tools was found by the author to be difficult. For example, data structure declarations generated by `pepsy` are used in the specification processed by `bison`. Changes to the conventions used by `pepsy` to name the generated data structures would require `a2b` to be changed to generate the correct `bison` specifications. Use of existing software tools also restricted the author's options for incorporating mechanisms such as debugging throughout the different parts of the generated interfaces.

The author's decision to generate interfaces that use communication protocols defined by international standards was made in the light of the growing trend in the field of geographical information systems to develop and use such standards. MACDIF and SDTS both use ISO standards to define parts of their data representation. The author used ISO standards ISO 8824, and ISO 8825, to define the network data representation used by a pair of communicating interfaces, and ISO 9072 to control the communication between a pair of communicating interfaces. The author found that gaining a working knowledge of these comprehensive and complex standards was a difficult and time-consuming process. However, the time spent gaining this knowledge was worthwhile because the author believes that using these standards allows data to be moved between a wide variety of computer systems.

While investigating the use of communications networks, the author concluded that there was a need to develop conventions, based on the use of existing standards, for governing the transfer of geographical data among interfaces linked by a communications network. These conventions ought to define at least the types of geographical data that may be communicated between the interfaces. The MACDIF definition (Evangalatos & Allam 1991) is one step in this direction. Other conventions may, for example, specify the remote operations that are to be used and/or provided by these interfaces.

To be effective, interfaces which use communication networks need to be efficient because they are likely to transfer data frequently. In the past, the time taken to transfer data has not been considered because of the delay in sending magnetic tapes from one computer system to another. More attention will have to be paid in the future to construction of interfaces that will operate efficiently.

Use of templates as the basis for the interfaces generated by `a2b` was found by the author to provided many advantages. Most significantly, different templates could be developed which contain a variety of functions that performed tasks other than data transfers. Thus, interfaces with a variety of functions could be constructed, without modifying `a2b` itself. Other advantages were connected with the ease with which programming errors within the generated interfaces could be corrected without needing to change `a2b`.

Use of `a2b` was intended for constructing interfaces that transfer geographical data. However, `a2b` can be used to construct interfaces that will transfer *any* type of data set between

two different text file representations. Therefore, the author believes that the concepts and methods described in this thesis have contributed to understanding and simplifying the construction of interfaces to transfer geographical and other types of data.

Overviews of ISO OSI standards

Appendix A

This Appendix is an overview of the following ISO-OSI standards: ISO 8211, used in the SDTS; ISO 8824, used in MACDIF; and ISO 8825, a standard closely associated with ISO 8824.

A.1 ISO 8211

ISO8211 (1985) specifies a medium-independent and system-independent file representation for transferring a wide variety of data types that may differ in size and complexity. Representations are defined for data of the following forms: indivisible data values, vectors, arrays, and hierarchies. A representation for network data structures is also defined, although 'additional pre-processing and post-processing is necessary to preserve logical linkages.' Three levels of representation are defined (ISO8211 1985):

level one

'supports multiple fields containing simple, unstructured character strings';

level two

'supports level one and processes multiple fields containing structured user data comprising a variety of data types';

level three

'supports level two and hierarchical data structures'.

The general structure of a data descriptive file is presented in Figure A.1 using a notation called Backus Naur Form (BNF), used by Naur *et al* (1963) to define the syntax of ALGOL 60.

A data descriptive file contains two types of record:

⟨data descriptive file⟩ ⟶ [⟨file labels⟩] ⟨Data Descriptive Record⟩
{⟨Data Record⟩} [⟨file labels⟩]
⟨Data Descriptive Record⟩ ⟶ ⟨DDR leader⟩ ⟨directory⟩ ⟨data descriptive area⟩
⟨DDR leader⟩ ⟶ ⟨record length⟩ ⟨interchange level⟩
⟨leader identifier⟩ ⟨inline code extension indicator⟩
⟨reserved⟩ ⟨application indicator⟩ ⟨field control length⟩
⟨base address of data descriptive area⟩
⟨extended character set indicator⟩ ⟨directory entry map⟩
⟨directory entry map⟩ ⟶ ⟨size of field length⟩ ⟨size of field position⟩
⟨reserved⟩ ⟨size of field tag⟩
⟨directory⟩ ⟶ {⟨directory entry⟩} ⟨field terminator⟩
⟨directory entry⟩ ⟶ ⟨field tag⟩ ⟨field length⟩ ⟨field position⟩
⟨data descriptive area⟩ ⟶ ⟨file control field⟩ ⟨field terminator⟩ {⟨data descriptive field⟩}
⟨data descriptive field⟩ ⟶ ⟨description of elementary data⟩
| ⟨description of compound data⟩
⟨description of elementary data⟩ ⟶ ⟨user data field name⟩
⟨description of compound data⟩ ⟶ 0 ⟨elementary data type⟩ ⟨reserved⟩ ;&
⟨description of elementary data⟩
| 1 ⟨compound data type⟩ ⟨reserved⟩ ;&
⟨description of vector data⟩
| 2 ⟨compound data type⟩ ⟨reserved⟩ ;&
⟨description of array data⟩
⟨elementary data type⟩ ⟶ ⟨character⟩ | ⟨implicit point⟩ | ⟨explicit point⟩
| ⟨explicit point, scaled⟩ | ⟨character mode bit string⟩ | ⟨bit field⟩
⟨compound data type⟩ ⟶ ⟨elementary data type⟩ | ⟨mixed types⟩
⟨description of vector data⟩ ⟶ [⟨user data field name⟩] ⟨unit terminator⟩
[⟨sequence of labels⟩] ⟨unit terminator⟩
[⟨format control⟩] ⟨field terminator⟩
⟨description of array data⟩ ⟶ [⟨user data field name⟩] ⟨unit terminator⟩ [⟨cartesian label⟩]
⟨unit terminator⟩ [⟨format control⟩] ⟨unit terminator⟩
[⟨array descriptor⟩] ⟨field terminator⟩
⟨cartesian label⟩ ⟶ {⟨sequence of labels⟩ ⟨delimitor⟩} ⟨sequence of labels⟩
⟨sequence of labels⟩ ⟶ {⟨label⟩ ⟨label delimitor⟩} ⟨label⟩
⟨Data Record⟩ ⟶ ⟨DR leader⟩ ⟨directory⟩ ⟨user data area⟩
⟨DR leader⟩ ⟶ ⟨record length⟩ ⟨reserved⟩ ⟨leader identifier⟩ ⟨reserved⟩
⟨base address of data area⟩ ⟨reserved⟩ ⟨directory entry map⟩
⟨user data area⟩ ⟶ ⟨record identifier⟩ ⟨field terminator⟩ {⟨user data field⟩}

Figure A.1: A BNF overview of the Data Descriptive File structure as specified by the ISO standard 8211. A more complete BNF description is given by van Roessel *et al* (1986)

a Data Descriptive Record (DDR)

which 'logically precedes the data records and contains the control parameters and data definitions necessary to interpret companion data records'(ISO8211 1985);

a Data Record (DR)

which contains representations of the data values according to the information given in the preceding data descriptive record.

Both types of record have a leader, which is 'a fixed length field that occurs at the beginning of each record and provides parameters for the processing of the record' (*op cit*). Parameters common to both record types are:

- a record length, specifying the total length of the record in bytes;
- a leader identifier, which is either the letter L in the case of a DDR, or either of the letters D or R in the case of a DR;
- a base address of either the data descriptive area of a DDR, or the user data area of a DR. The base address for either data area indicates the position of the first data field in the area; and
- a directory entry map, describing the entries in the directory field.

A DDR leader field consists of an additional parameter, the interchange level, indicating whether the file conforms to level 1, 2, or 3 of the interchange specification.

Both a DDR and a DR have a directory field after the leader field. Entries in the DDR directory describe all fields in the data descriptive area, including the file control field. Similarly, entries in the DR directory describe all fields in the user data area, including the record identifier field. Entries in both the DDR and DR directories consist of:

- a field tag, which uniquely identifies either a data descriptive field in the case of a DDR, or a data field in the case of a DR;
- a field length, which specifies the number of bytes in the field identified by the field tag; and
- a field position, which is the position of the first byte for this field in the data area, relative to the base address of the data area within a record.

The data descriptive area of a DDR consists of fields that describe the fields in the associated DRs. The data descriptive fields in a DDR are of one of two types because fields of a DR may be either elementary data fields, in which an indivisible data values are

represented, or compound data fields, in which a one or more data values may be organised. The two types of DDR data descriptive fields are therefore:

- an elementary data description field, consisting of the name of the corresponding user data field, or
- a compound data description field, consisting of four parts:
  - a one digit code indicating whether the data is an elementary data value (0), a vector data value(1), or an array data value (2);
  - a one digit code indicating which of six types of data is represented (0 – character, 1 – implicit point, 2 – explicit point, 3 – explicit point, scaled, 4 – character mode bit string, 5 – bit field, 6 – mixed types);
  - a name for the described data field; and
  - a collection of labels and format controls in a form corresponding to the structure and type of values represented.

The user data area of a DR consists of a record identifier field, and one or more user data fields that are structured according to the corresponding data descriptive field in the DDR. This correspondence is indicated by the directory entries for the data descriptive record and corresponding user data field having the same field tag. Given the wide variety of user data fields, the user data area of a DR is explained by the example shown in Figure A.2.

```
001963L   0600060   2304
000037000000130037ØBID34067SPAD35101
0000;&Example data descriptive file&;
0100;&DDF Record Identifier&&;
0100;&spatial object identifier&&;
2600;&spatial address&*LNG!LAT&(I);
00077 D     00051   2304
0001060000BID04006SPAD16010
00001;345;423&132&426&135;
```

Figure A.2: An example of a data descriptive file. The symbol ';' is used to mark a field terminator and the symbol '&' is used to indicate a unit terminator

Figure A.2 is an example of a data descriptive file in which one line segment is represented. The line segment consists of an identifier `345` and the two end points `(423,132)` and `(426,135)`. For clarity, the content of the file is broken into lines, the character '&' is used to represent a unit terminator, and the character ';' is used to represent the field terminator. The Data Descriptive Record (DDR) is spread across the first 6 lines, with the DDR leader shown on the first line, the directory shown on the second, and the data descriptive area

shown on lines 3–6. The Data Record (DR) is spread across the remaining 3 lines, with the DR leader shown on line 7, the directory shown on line 8, and the user data area shown on line 9.

The DDR consists of `00196` characters, with the data descriptive area starting at character `00060`. The directory entry map `2304` indicates that each entry will consist of 9 characters: the first 4 contain the data descriptive field tag, the next 2 specify the length of this data descriptive field, and the remaining 3 characters indicate the position of this data descriptive field relative to the start of the data descriptive area. The third directory entry `SPAD35101`, for example, indicates that the data descriptive field occupying 35 characters starting at character 101 is identified by the tag `SPAD`.

The data descriptive field starting at character `101` describes a user data field that has a name of `spatial address`. Data in this user data field consists of an array with elements of mixed data types, indicated by the file control field consisting of `2600;&`. The array has two columns, longitude with the label `LNG` and latitude with the label `LAT` containing the longitude and latitude values of the line segment.

The third entry in the data record directory `SPAD16010` indicates that the user data field starting at character `010` relative to the start of the user data area is to be interpreted according to the data descriptive field associated with the tag `SPAD`. That is, the third field in the user data area consisting of `423&132&426&135` is an array of spatial addresses, having two rows corresponding to the end points of the line segment.

A.2 ISO 8824

The definition of ASN.1 specifies a set of *simple types* such as:

Integer

'a simple type with distinguished values which are the positive and negative whole numbers, including zero (as a single value)' (ISO8824 1987, clause 3.16). The notation for this type is the keyword `INTEGER`;

Boolean

'a simple type with two distinguished values' (clause 3.13, *op cit*). The notation for this type is the keyword `BOOLEAN`, and the two values are the keywords `TRUE` and `FALSE`;

Octet String

'a simple type whose distinguished values are an ordered sequence of zero, one or more octets, each octet being an ordered sequence of eight bits' (clause 3.18, *op cit*). Notation for this type are the keywords `OCTET STRING`.

A *structured type* is some combination of simple types and/or previously defined, structured types. ASN.1 provides a variety of methods for defining a structured type, including:

- 'given a single existing type, a value can be formed as an (ordered) sequence or (unordered) set of zero, one or more values of the existing type; the (infinite) collection of all possible values obtained in this way is a new type' (ISO8824 1987, page 2). A new structured type, called `SequenceOfPoints`, consisting of an (ordered) sequence of `point` (a previously defined structured type) is defined as follows:

```
SequenceOfPoints ::= SEQUENCE OF Point
```

- 'given a list of (distinct) types, a value can be chosen from any one of them; the set of all possible values obtained in this way is a new type' (page 2, *op cit*). For example, points may be defined in either two or three dimensions. This may be defined as follows:

```
GeneralPoint ::= CHOICE {
    twoDpoint   [ 0 ] A2dPoint,
    threeDpoint [ 1 ] A3dPoint
    }
A2dPoint ::= SEQUENCE {
    x [0] INTEGER,
    y [1] INTEGER
    }
A3dPoint ::= SEQUENCE {
    x [0] INTEGER,
    y [1] INTEGER,
    z [3] INTEGER
}
```

  A value of the type `GeneralPoint` will be of either a `A2dPoint` type, or a `A3dPoint` type. Although the identifiers `twoDpoint` and `threeDpoint` in the above specification must be unique within the `CHOICE` (and any other similar) construction, the tags `[ 0 ]` and `[ 1 ]` serve to distinguish between the types that may be chosen. Tags may be used to distinguish between two occurrences of the same type, allowing a choice between two occurrences of the `INTEGER` type, for example.

Every type defined using ASN.1 is assigned a *tag*, belonging to one of four classes (ISO8824 1987):

Universal

Universal class tags are assigned to either single types, or construction mechanisms

such as a `CHOICE` or a `SEQUENCE OF`. Universal tags assignments are specified by the ISO standard 8824;

Application

Application class tags are assigned to types by other standards;

Private

Private class tags are assigned for use within an application and are never assigned by international standards; and

Context-specific

Context-specific class tags are 'freely assigned within any use of ASN.1, and is interpreted according to the context in which the tag is used.

An example of an abstract syntax defining the type `PointList` using ASN.1 is given in Figure A.3, and in Table A.1 a representation is given for the value $5, 7, 9, 8$ of type `PointList` according to the encoding rules specified by ISO 8825.

```
EX DEFINITIONS ::=
BEGIN
PointList ::= [PRIVATE 1] SEQUENCE OF Point

Point ::= [2] SEQUENCE {
                              x [0] INTEGER,
                              y [1] INTEGER
          }
END
```

Figure A.3: An example of an abstract syntax expressed using Abstract Syntax Notation One (ASN.1)

A.3 ISO 8825

ISO 8825 specifies some basic encoding rules which when applied to values of types defined using ASN.1 produces a physical representation of these values. The encoding of any value is defined using BNF as follows:

⟨value encoding⟩ ⟶ ⟨identifier octets⟩ ⟨contents⟩

⟨contents⟩ ⟶ ⟨definite length octets⟩ ⟨contents octets⟩

| ⟨indefinite length octets⟩ ⟨contents octets⟩ ⟨end-of-content octets⟩

where

Table A.1: Representation of the data value {{5,7},{9,8}} of the type `PointList` defined by the abstract syntax given in Figure A.3. Use of the ASN.1 keyword `IMPLICIT` in this abstract syntax would have produced a more compact representation

`PointList` data value {{5,7},{9,8}}		
ISO 8825 representation: $E11E301CE20C300AA003020105A103020107E20C300AA003020109A103020108_{16}$		
Tag: [P 1]	Lgth	Contents
$\underbrace{11100001}_{E1_{16}}$	$1E_{16}$	$301CE20C300AA003020105A103020107E20C300AA003020109A103020108_{16}$

Tag: Sq-of	Lgth	Contents
$\underbrace{00110000}_{30_{16}}$	$1C_{16}$	$E20C300AA003020105A103020107E20C300AA003020109A103020108_{16}$

Tag: [P 2]	Lgth	Contents
$\underbrace{11100010}_{E2_{16}}$	$0C_{16}$	$300AA003020105A103020107_{16}$

Tag: Seq	Lgth	Contents
$\underbrace{00110000}_{30_{16}}$	$0A_{16}$	$A003020105A103020107_{16}$

Tag: [0]	Lgth	Contents
$\underbrace{10100001}_{A0_{16}}$	03_{16}	020105_{16}

Tag: Integer	Lgth	Contents
$\underbrace{00000010}_{02_{16}}$	01_{16}	$\underbrace{00000101}_{05_{16}}$

Tag: [1]	Lgth	Contents
$\underbrace{10100001}_{A1_{16}}$	03_{16}	020107_{16}

Tag: Integer	Lgth	Contents
$\underbrace{00000010}_{02_{16}}$	01_{16}	$\underbrace{00000111}_{07_{16}}$

Tag: [P 2]	Lgth	Contents
$\underbrace{11100010}_{E2_{16}}$	$0C_{16}$	$300AA003020109A103020108_{16}$

Tag: Seq	Lgth	Contents
$\underbrace{00110000}_{30_{16}}$	$0A_{16}$	$A003020109A103020108_{16}$

Tag: [0]	Lgth	Contents
$\underbrace{10100000}_{A0_{16}}$	03_{16}	020109_{16}

Tag: Integer	Lgth	Contents
$\underbrace{00000010}_{02_{16}}$	01_{16}	$\underbrace{00001001}_{09_{16}}$

Tag: [1]	Lgth	Contents
$\underbrace{10100001}_{A1_{16}}$	03_{16}	020108_{16}

Tag: Integer	Lgth	Contents
$\underbrace{00000010}_{02_{16}}$	01_{16}	$\underbrace{00001000}_{08_{16}}$

identifier octets

encode the class and number of the tag associated with the type of data value, and indicate the encoding technique used for the contents octets (described below);

length octets

may be of one of two forms:

definite form

where the number of content octets is encoded in one or more length octets, or

indefinite form

where the contents of the length octet indicates that the content octets are terminated by some end-of-contents octets;

contents octets

encode a value of the type indicated by the tag contained within the identifier octets, according to the encoding rules specified by ISO 8825 (ISO8825 1987). Two examples of these rules are briefly described here:

the primitive encoding of an integer value

comprising one or more octets that form a twos-complement binary number equal to the integer value; and

the constructed encoding of a sequence-of value

comprising zero, one or more complete encodings of values of the type specified by the corresponding ASN.1 syntax. Each complete encoding of a value will have identifier octets, length octets, contents octets, and, if indicated by the length octet, end-of-contents octets;

end-of-contents octets

consisting of two zero octets. The presence of the end-of-contents octets is optional and is indicated by the content of the length octets.

Early and present approaches to interface construction

Appendix B

In this Appendix, three different interfaces are presented to perform the same data transfer $\mathcal{S} \xrightarrow{*} \mathcal{D}$, where the data set $\mathcal{S}$ conforms to a simplified Gina data file format, and $\mathcal{D}$ conforms to a data file representation similar to that required by the geographical information system Grass. Examples of the data sets $\mathcal{S}$ and $\mathcal{D}$ are shown in Figure B.1.

```
feat 874 L
coor 867 543 874 550 880 564
feat 875 L
coor 880 564 876 580
feat 876 P
coor 872 550
```

The source data set $\mathcal{S}$

```
Line 874 3 867 543 880 564
867 543
874 550
880 564
Line 875 2 876 564 880 580
880 564
876 580
Point 876 872 550
```

The destination data set $\mathcal{D}$

Figure B.1: Example data sets

The source code for the first of these interfaces is presented in Section B.1. This source code must be constructed by hand when using the approach to interface construction developed by the author in earlier research which was described in Section 5.2.4. The second and third interfaces are generated by `a2b` from transfer specifications presented in Section B.2.

The purpose of this Appendix is to demonstrate how much simpler the approach to interface construction described in this thesis is in comparison with the approach developed in earlier research.

B.1 Interface constructed using earlier approach

B.1.1 Main program

```
#include <stdio.h>
#define dupstr(A,B) {\
    if (A) { \
        *(B) = (char *) malloc(strlen(A)+1); strcpy(*(B),A);\
    } else\
        *(B) = NULL;\
    }

extern char *optarg;
extern int optind, createSrcDb(), openDb(), destroySrcDb(), closeDb();

main(argc, argv)
        int argc;
        char **argv;
{
      char *srcFile = NULL, *destFile = NULL;

      doOptions(argc, argv, &srcFile, &destFile);
      openDb("eg1"); createSrcDb();
      decode(srcFile);
      translate();
      encode(destFile);
      destroyDestDb(); destroySrcDb();
      closeDb("eg1");
 }

doOptions(argc, argv, srcFilep, destFilep)
     int argc;
     char **argv, **srcFilep, **destFilep;
{
      int c;
      optind = 1;
      while (( c = getopt(argc, argv, "o:")) != -1)
            switch(c) {
               case 'o' :
```

```
                dupstr(optarg, destFilep);
                break;
            default:;
        }
    dupstr(argv[optind], srcFilep);
}
```

B.1.2 Decoder module

This listing of the decoder module is divided into three parts: the yacc specification file; the lex specification file; and the routines for accessing an Ingres database.

Yacc specification file

```
%{
#include <stdio.h>
extern appendFeature(), appendCoor();
%}
%pure_parser
%union{ char *string;   int integer; }
%token FEAT COOR
%token <string> STRING
%token <integer> INTEGER
%%
file        : feature | file feature ;
feature     : FEAT description COOR coordinates ;
description : INTEGER STRING { appendFeature( $1, $2); };
coordinates : INTEGER INTEGER { appendCoor($1, $2); }
            | coordinates INTEGER INTEGER { appendCoor($2, $3); };
%%
#include "decoderLex.c"

decode(filename) char *filename;
{
    FILE *freopen();

    freopen(filename, "r", stdin);
    yyparse();
}
yyerror(s) char *s;
```

```
{
    fprintf(stderr, "parse error with string %s\n", yytext);
}
```

Lex specification file

```
%{
#undef YY_DECL
#define YY_DECL static int yylex(yylvalp,yyllocp) YYSTYPE *yylvalp; YYLTYPE *yyllocp;
#define dupstr(A,B) {\
    if (A) { \
        *(B) = (char *) malloc(strlen(A)+1); strcpy(*(B),A);\
    } else\
        *(B) = NULL;\
    }

int nlines =0;
%}
D               [0-9]
CHAR            [A-Za-z]
%%
feat                { return FEAT; }
coor            { return COOR; }
{CHAR}+         { dupstr(yytext,&((*yylvalp).string)); return STRING; }
{D}+            { (*yylvalp).integer = atoi(yytext); return INTEGER; }
.|\n                /* ignore */
```

Database routines

```
#include <stdio.h>
##  int featureId, coorSeq;

openDb(dbname)
##  char *dbname;
{
##  ingres dbname
}
closeDb(dbname) char *dbname;
{
##  exit
```

```
}
createSrcDb()
{
##  create feature(id = i2, ftype = c2)
##  create coor(id = i2, seq = i2, x = i2, y = i2)
}
destroySrcDb()
{
##  destroy feature,coor
}
destroyDestDb()
{
##  destroy linedescription, linedefn, point
}
appendFeature(id, featType)
    int id;
##  char *featType;
{
    featureId = id; coorSeq = 0;
##  append feature(id = featureId, ftype = featType)
}

appendCoor(xVal, yVal)
##  int xVal, yVal;
{
##  repeat append coor(id = @featureId, seq = @coorSeq, x = @xVal, y = @yVal)
    coorSeq++;
}
```

B.1.3 Translator module

```
#include<stdio.h>

translate()
{
##  retrieve linedefn(id = coor.id, seq = coor.seq, x = coor.x, y = coor.y)
##    where coor.id = feature.id and feature.ftype = "L"
##  retrieve point(id = coor.id,x = coor.x, y = coor.y)
##    where coor.id = feature.id and feature.ftype = "P"
```

```
##  retrieve linedescription(
##                  id = coor.id, npts = count(coor.seq by coor.id),
##                  mnx = min(coor.x by coor.id), mny = min(coor.y by coor.id),
##                  mxx = max(coor.x by coor.id), mxy = max(coor.y by coor.id))
##      where count(coor.seq by coor.id) > 1
}
```

B.1.4 Encoder module

```
#include<stdio.h>

struct LineDescription {
    int id, npts, mnx, mny, mxx, mxy;
    struct LineDescription *next;
} * ld;

##  int  s, xv, yv, lid, lnpts, lmnx, lmny, lmxx, lmxy;

encode(filename) char *filename;
{
    freopen(filename, "w", stdout);
    doLines();
    doPoints();
}

doLines()
{
    getLineDescriptions(&ld);
    while(ld) {
        lid = ld->id;
        printf("Line %d %d %d %d %d %d\n", ld->id, ld->npts, ld->mnx, ld->mny,
               ld->mxx, ld->mxy);
##      retrieve ( lid = linedefn.id, s = linedefn.seq,
##                  xv = linedefn.x, yv = linedefn.y)
##         where linedefn.id = lid
##       sort by #lid, #s
##{
           printf("%d %d\n", xv, yv);
##}
```

```
        ld = ld->next;
    }
}
doPoints()
{
##  retrieve (lid = point.id, xv = point.x, yv = point.y)
##{
        printf("Point %d %d %d\n", lid, xv, yv);
##}
}
getLineDescriptions(hdp) struct LineDescription **hdp;
{
    struct LineDescription *hd = NULL, *tl = NULL, *tmp = NULL;

##  retrieve(lid = linedescription.id, lnpts = linedescription.npts,
##           lmnx = linedescription.mnx, lmny = linedescription.mny,
##           lmxx = linedescription.mxx, lmxy = linedescription.mxy)
##{
        tmp = (struct LineDescription *) malloc(sizeof(*tmp));
        tmp->id = lid; tmp->npts = lnpts; tmp->mnx = lmnx; tmp->mny = lmny;
        tmp->mxx = lmxx; tmp->mxy = lmxy; tmp->next = NULL;

        if (hd && tl)
            tl->next = tmp;
        else
            hd = tmp;
        tl = tmp;
##}
    *hdp = hd;
}
```

B.2 Interface constructed using current approach

In this Section, two transfer specifications for the same data transfer are presented for comparison with the approach described in Section B.1. The transfer specification presented in Section B.2.1 includes a translator module specified using the Miranda translation environment. The specification presented in Section B.2.2 includes a translator module specified using the C translation environment. Although these specifications have different translator

modules, they are equivalent in the sense that the interfaces generated by a2b from these specifications perform the same data transfer.

B.2.1 Transfer specification with a Miranda translator module

```
%a2b source to destination

%source
source: (feature || "\n" )* ;
feature: ("feat" (id type) "\ncoor" (point)*);
point:(x y);
id, x, y:integer;
type:<< "L" | "P" >>;

%destination
destination: (
   < "Line " (( id #nPts minBoundRect) "\n"
              (point || "\n")^nPts)
   | "Point " ( id x y )> "\n" )*;
point:(x y);
minBoundRect:(mnx mny mxx mxy);
id, nPts, x, y, mnx, mny, mxx, mxy:integer;

%source2destination %mira{
source2destination data
  = map src2destObj data
    where
      src2destObj ((id,0),pts)
        = Destination_choice_1 ((id, # pts, mbr), pts)
          where
            thePts = foldl dopts ((10000, 10000, -10000, -10000), []) pts
            mbr = extract_mbr thePts where extract_mbr (a,b) = a
      src2destObj ((id,1),(x,y):pts)
        = Destination_choice_2 (id, x, y)
      dopts ((mnx, mny, mxe, mxn), xys) (x,y)
          = ((min2 x mnx, min2 y mny, max2 x mxe, max2 y mxn), (x,y):xys)
%}
```

B.2.2 Transfer specification with a C translator module

```
%a2b source to destination
%source
source:       (feature || "\n" )* ;
feature:      ("feat" (id type) "\ncoor" (point)* );
point:        (x y);
id, x, y:     integer;
type:         << "L" | "P" >>;

%destination
destination: ( < "Line " (( id #nPts minBoundRect) "\n" (point || "\n")^nPts)
               | "Point " ( id x y )> "\n")*;
point:        (x y);
minBoundRect:(mnx mny mxx mxy);
id, nPts, x, y, mnx, mny, mxx, mxy:integer;

%source2destination %c{
#define new(A,B) *A = (struct type_DESTINATION_/**/B *) malloc(sizeof(**A));
#define setNext(A,B,C) if (A) { new(B,C); (*B)->next = NULL; } else (*B) = NULL;
#define Sdecl(A,B) struct type_SOURCE_/**/A B
#define Ddecl(A,B) struct type_DESTINATION_/**/A B
#define Dfeature d->element_DESTINATION_0
#define Sfeature s->element_SOURCE_0
#define Spoint Sfeature->casS2->element_SOURCE_1
#define Dpoint Dfeature->un.casS6
#define Dline Dfeature->un.casS4
#define DlineInfo Dline->casS0
#define Dmbr DlineInfo->elm0
#define setMinMbr(A,B) \
    Dmbr->A = (init || Dmbr->A > sP->element_SOURCE_1->B ? \
               sP->element_SOURCE_1->B : Dmbr->A);
#define setMaxMbr(A,B) \
     Dmbr->A = (init || Dmbr->A < sP->element_SOURCE_1->B ? \
               sP->element_SOURCE_1->B : Dmbr->A);

source2destination(s,dp) Sdecl(Source, *s); Ddecl(Destination, **dp);
{
      Ddecl(Destination, *d); Sdecl(CasS2, *sP); Ddecl(CasS2, *dP);
```

```
ew(&d,Destination);*dp = d;
or (;s;s=s->next,d=d->next) {
     new(&Dfeature,CasS8);
     switch (Sfeature->casS0->type) {
        case 0 : { /* Line */
              int init = 1;
              Dfeature->offset = type_DESTINATION_CasS8_casS4;
              new(&Dline,CasS4); new(&DlineInfo,CasS0);
              DlineInfo->id = Sfeature->casS0->id;
              DlineInfo->nPts = 0; DlineInfo->elm0 = NULL;
              if (Sfeature->casS2) {
                    new(&Dline->casS2,CasS2);
                    new(&DlineInfo->elm0,MinBoundRect);
                    for (sP = Sfeature->casS2, dP = Dline->casS2; sP;
                         sP=sP->next, dP=dP->next,init = 0) {
                       DlineInfo->nPts++;
                       new(&dP->member_DESTINATION_0,Point);
                       dP->member_DESTINATION_0->x = sP->element_SOURCE_1->x;
                       dP->member_DESTINATION_0->y = sP->element_SOURCE_1->y;
                       setMinMbr(mnx,x); setMinMbr(mny,y);
                       setMaxMbr(mxx,x); setMaxMbr(mxy,y);
                       setNext(sP->next,&dP->next,CasS2); } }
              break; }
        case 1 : { /* Point */
              Dfeature->offset = type_DESTINATION_CasS8_casS6;
              new(&Dpoint,CasS6);
              Dpoint->id = Sfeature->casS0->id;
              Dpoint->x = Spoint->x; Dpoint->y = Spoint->y;
              break; }}
     setNext(s->next,&d->next,Destination); }
```

UNIX on-line manual pages

Appendix C

kerngen

NAME

kerngen – generate kerning table from a lexical analyser specification

SYNOPSIS

```
kerngen [-sTD] [flex-output-file] [-r state] [-t token-file]
[-x extra-file] [-o output-file] [-L lib-dir]
```

OPTIONS

flex-output-file	Input is from the named flex output file, or stdin if no input file is specified.
-r number	Specifies the "root state" from which to begin. Kerngen will only consider states reachable from this state. The default is state 0 (the initial start state).
-t token-file	Specifies the name of an file containing token definitions which will be included in the generated C file.
-x extra-file	Specifies a file of extra kerning specifications (see below).
-o output-file	Specifies the name of the generated C file. If no output file is specified, output is written to stdout.
-s	Generate a stand-alone output file which can be compiled without needing any additional files.
-L lib-dir	Specifies the directory in which the standard files "kerning.h" and "kerning.c" may be found. Defaults to the current directory.
-T	Debugging option. Write to stderr the tables extracted from the input file.
-D	Debugging option. Write to stderr a representation of the DFA constructed from the input tables.

DESCRIPTION

Kerngen is a tool for use in the creation of "unparsers" which generate output in the form of a sequence of lexical tokens, where the tokens are defined by a flex(1) specification.

When generating output of this sort, it is usually necessary to include whitespace between some pairs of tokens, so that the resulting character sequence will not be mis-interpreted. For instance, in a typical programming language, two identifiers cannot be placed together without space between, lest they become another single identifier.

The naive approach of inserting space between every pair of tokens, while it usually produces correct output, does not generally produce results which are aesthetically pleasing to a human reader.

Kerngen is designed to aid in the production of tidier output by generating a *kerning table* for token pairs. The term "kerning" is borrowed from typography, where it means adjusting the amount of space between pairs of letters to achieve a pleasing appearance. Here, it means the inclusion or omission of whitespace between pairs of tokens to achieve a result which is foremost unambiguous, and secondmost pleasing.

The input to kerngen is the finite state machine generated by flex from the flex specification of the tokens. The output from kerngen is a kerning table containing an entry for each pair of tokens which require separation by whitespace.

Accompanying the kerning table is a *kerning function* which takes a pair of tokens as parameters, looks up the table, and determines whether whitespace should be inserted between them. The kerning table and kerning function are described in more detail below.

Algorithm

The algorithm used by kerngen to calculate the kerning table is as follows. Each state *f1* which can serve as a final state returning token *t1* is examined. If there is a transition from *f1* on character *c*, and there is also a transition from the initial state *s0* on *c* to state *s*, then each final state *f2* reachable from *s* by some sequence of of transitions is found.

If *f2* returns *t2*, then there is considered to be a possible conflict between *t1* and *t2*. That is, if *t1* and *t2* were to be concatenated without intervening space, ambiguity could result. An entry is therefore made in the kerning table containing *t1* and *t2*.

This algorithm is conservative; it is possible that a conflict between two tokens will be reported even though no such conflict actually exists. The algorithm could be made exact by following transitions from *f1* and *s0* in parallel until both paths arrive at a final state simultaneously before reporting a conflict. In many practical cases, however, the current algorithm gives the same results as the exact algorithm.

Input

Kerngen takes as input the lex.yy.c file generated by flex(1). Two other pieces of information may be supplied: the name of an file containing the definitions of token names (generated by yacc(1) or bison(1) using the -d option); and a file containing additional kerning specifications (see below).

Output

The output is a C file containing a data structure which is used by the kerning function. This file may either be compiled separately, or included in another C file.

If the -s option is used, a stand-alone output file is generated, containing the kerning table and kerning function together with any necessary declarations. This file can be compiled and used without requiring any additional files.

Without -s, an output file is generated containing the kerning table only. To compile this file, the file "kerning.h" must be available, together with any file specified in the -t option. Also, the file "kerning.c" must be compiled and linked with the final program.

Kerning Function

The kerning function is defined in the file "kerning.c". The kerning function is declared as

```
char *kerning(t1, t2)
TOKEN t1
```

The type TOKEN, and the type of the kerning table structure, are declared in "kerning.h". This file also declares the variable

```
extern char *default_whitespace;
```

which should be defined elsewhere to point to the default separating string.

The return value is a separating string to be emitted between the two tokens, or NULL if no separator is required.

Additional Kerning Specifications

The kerning table generated by default specifies the minimum amount of whitespace necessary to avoid ambiguity. In some cases it is desirable to additional

whitespace; for instance in a programming language it might be desirable to insert a newline after every semicolon.

For this purpose a file of additional kerning specifications may be supplied. The file should contain lines of the form

```
<token1> <token2> <sep>
```

where ¡token1¿ and ¡token2¿ are token names, and ¡sep¿ is a C string constant containing whitespace. Zero may be used instead of a token to mean "any token", and instead of a separator to mean "default whitespace". (Not all three fields may be zero simultaneously.)

FILES

```
kerning.h Kerning table data structure declarations
kerning.c Definition of kerning function
```

SEE ALSO

flex(1), yacc(1), bison(1)

AUTHOR

Gregory C. Ewing, University of Canterbury

BUGS

Kerngen is fussy about the textual form of the "return" statement in an action returning a token.

Flex seems to generate differing forms of output tables depending on the nature of the flex specification in a way that is not well understood. Kerngen at present only understands one of these forms, and fails if it is given any of the others.

Any change in the flex output format is likely to render kerngen inoperative.

Kerngen should read the flex specification directly instead of relying on flex.

a2b

NAME

a2b – generate one or more interfaces from a transfer specification

SYNOPSIS

```
a2b [-vtnl] [-d < scanner | parser | decoder | responder > ]
    [-S template-directory]
    transfer-specification
```

OPTIONS

-v Debugging option. Write to standard error the command lines for invoking the different software tools used to produce the interface.

-t An option to indicate that temporary files are to be removed after use.

-n Debugging option. Instructs the software tool **a2b** not to invoke any of the other software tools used to generate parts of an interface.

-l Create the file **a2b.log** which contains any messages that are produced by other software tools while generating parts of an interface.

-S *template-directory*
Specifies the directory in which are located the templates to be used by the software tool **a2b** for generating any interfaces.

-d *level* Debugging option. Specifies the level of debugging to be included within the generated interfaces, where *level* may be one of the following:

scanner All debugging facilities provided by the software tool **flex** are included within the scanner generated from each representation definition within a transfer specification;

parser All debugging facilities provided by the software tool **bison** are included within the parser generated from each representation definition within a transfer specification;

decoder All debugging facilities for both the scanner and parser are included for every decoder module within the generated interface;

responder All debugging facilities provided by **a2b** are inserted into the responder. This is essentially an infinte loop to be broken using a source-level debugger; and

transfer-specification

Specifies the name of the file containing the transfer specification to be processed.

DESCRIPTION

The software tool **a2b** generates *interfaces* to perform a data transfer according to some transfer specification. In the process of transferring the data, one or more interfaces transform the *source data set* conforming to some representation A, into the *destination data set* conforming to some other representation B. During the transfer, data comforming to some network representation may also be moved from one computer system to another by these interfaces using the implementation of various ISO-OSI communication protocols provided by ISODE (Rose, 1991).

The software tool **a2b** will generate either a single interface, which does not move data from one computer system to another, or a pair of communicating interfaces. The *source communicating interface* transforms the source data set into another data set conforming to some network representation. The *destination communicating interface* transforms the set of data values conforming to some network representation into the destination data set.

The software tool **a2b** generates parts of the interfaces in different ways. Some parts of the interfaces are generated by other software tools processing specifications produced by the software tool **a2b**, some parts are generated directly by the software tool **a2b**, and the remainder is provided in different interface templates.

In Pascoe(1994) a more detailed discussion is provided of the different interfaces generated by the software tool **a2b**, the software tool **a2b** itself, and the notation for expressing transfer specifications to be processed by the software tool **a2b**.

SEE ALSO

Pascoe (1994), *Construction of interfaces for the transfer of data between geographical information systems*, Doctorate thesis, Dept. of Computer Science, University of Canterbury.

Rose, M. T., Onions, J. P. and Robbins, C. J. (1991), *The ISO Development Environment: Users Manual*, 7th Ed., Performance Systems International and X-Tel Services.

AUTHOR

Richard T. Pascoe, University of Canterbury.

imisc

NAME

imisc – miscellaneous network service – initiator

SYNOPSIS

```
imisc                host                [ prefix arguments ]
     [operation [ arguments ... ] ]
```

DESCRIPTION

The **imisc** program requests an operation from the miscellaneous network service provided by a server using remote operations services. The currently supported operations are:

operation	description
utctime	the universal time
gentime	the generalized time
time	the current time since the epoch
users	the users logged in on the system
chargen	the character generation pattern
qotd	the quote of the day
finger	the finger of users logged in
pwdgen	some pseudo-randomly generated passwords
tell	send a message to a remote user
ping	ping test for performance measurement
sink	sink test for performance measurement
echo	echo test for performance measurement

This program initiates remote operations to the 'isode miscellany' service.

The prefix arguments are used to direct how many times the performance measurement tests should run: the -c count switch indicates the number of iterations to try, and the -l length switch indicates the number of octets to send as user data for each iteration.

If no operation is given on the command line, then **imisc** enters interactive mode: **imisc** will examine each line of the standard-input, treating the first word as the operation, and any remaining words as arguments. (Currently, only the *finger* operation takes arguments.)

The pseudo-operations *help* and *quit* do the obvious things.

FILES

isode/etc/isoentities	ISODE entities database
isode/etc/isobjects	ISODE objects database
isode/etc/isoservices	ISODE services database

DIAGNOSTICS

Obvious.

AUTHOR

Marshall T. Rose.

Notation processed by a2b

Appendix D

In this Appendix is presented the bison grammar from which is generated the parser for the software tool a2b. Also included in this Appendix is a description of the differences between the notation defined in Chapter 7, and the notation processed by the software tool a2b.

D.1 Parser definition for a2b

In this Section is presented the formal definition of the a2b parser. This definition is divided into two parts. The first part, presented in Section D.1.1, is the grammar which is processed by the software tool bison to produce the parser. Although the actions are excluded from this listing, their position is marked by the string {...}. The second part, presented in Section D.1.2, is the scanner definition which is processed by the software tool flex to produce a scanner.

D.1.1 bison grammar

```
%{
/*
 *      Variable declarations and macro definitions used in the actions
 *      of the grammar
 */
%}

%pure_parser
%union {
        struct ISPEC *intDfn;          struct Module *module;
        struct AType *aType;           char *string;
```

```
        struct CAS *cas;            int intValue;
        struct TmpExp *anExp;       struct ModRef *modRef;
}
%type <intDfn> interface intDfn
%type <aType> cas choice identList rule irns litSeq aSeq litList tRef tDef ssco
%type <aType> litSet aSet literal type conRep primType aType typeList aTypeList
%type <aType> aLiteral litChoice anOption options scanSwtch nxtType sepClause number
%type <string> aToken
%type <modRef> modName
%type <anExp> exp

%token <string> IDENT LITERAL INTEGER REAL STRING MODULENAME WSPACE TRNSCODE
%token <intValue> INTVALUE
%token DFWSP DEFWSP ANYTKN LSQB RSQB LCB RCB LBRACE RBASTERISK RBPLUS RCASTERISK
%token RCPLUS RBRACE LAB RAB LAB2 RAB2 SEMI COLON BAR COMMA SETFLEX SREP CNTDECL
%token MIRAENV CENV GOFERENV A2B SEPBARS RLITERAL RCN RBN TO VIA
%start interface
%left MINUS PLUS
%left ANYTKN DIVI
%%
interface:      A2B intDfn {...} moduleDfns
        ;
intDfn  :       modName TO modName                      {...}
        |       modName VIA modName TO modName          {...}
        |       intDfn TO modName                       {...}
        |       intDfn VIA modName TO modName           {...}
        ;
modName :       IDENT                                   {...}
        ;
moduleDfns:     module | moduleDfns module
        ;
module:         MODULENAME scanSwtch options cas        {...}
        |       MODULENAME MIRAENV TRNSCODE             {...}
        |       MODULENAME CENV TRNSCODE                {...}
        |       MODULENAME GOFERENV TRNSCODE            {...}
        ;
options :                                               {...}
        |       options anOption                        {...}
        ;
```

```
anOption:       DFWSP LITERAL                          {...}
        |       DEFWSP LITERAL                         {...}
        |       DEFWSP aToken aToken LITERAL           {...}
        ;
aToken  :       INTEGER                                {...}
        |       REAL                                   {...}
        |       literal                                {...}
        |       STRING                                 {...}
        |       ANYTKN                                 {...}
        ;
cas     :               rule                           {...}
        |       cas     rule                           {...}
        ;
identList:      IDENT                                  {...}
        |       identList COMMA IDENT                  {...}
        ;
rule    :       identList COLON type SEMI              {...}
        |       identList COLON litList SEMI           {...}
        ;
type    :       aType                                  {...}
        |       aType litList                          {...}
        ;
aType   :       tRef                                   {...}
        |       tDef                                   {...}
        ;
tRef    :       irns                                   {...}
        |       litList irns                           {...}
        ;
tDef    :       ssco                                   {...}
        |       litList ssco                           {...}
        ;
typeList:       aTypeList                              {...}
        |       aTypeList litList                      {...}
        ;
aTypeList:      aType aType                            {...}
        |       aTypeList aType                        {...}
        ;
litList :       aLiteral                               {...}
        |       litList aLiteral                       {...}
```

```
        ;
aLiteral:       literal                                 {...}
        |       litSeq                                  {...}
        |       litSet                                  {...}
        |       LAB litChoice RAB                       {...}
        |       LSQB litList RSQB                       {...}
        |       WSPACE                                  {...}
        ;
literal :       LITERAL scanSwtch                       {...}
        ;
irns    :       CNTDECL IDENT scanSwtch                 {...}
        |       IDENT  scanSwtch                        {...}
        |       conRep scanSwtch                        {...}
        ;
conRep  :       primType                                {...}
        |       primType SREP LITERAL COLON LITERAL RCB {...}
        |       primType COLON INTVALUE                 {...}
        |       primType COLON IDENT                    {...}
        ;
primType:       INTEGER                                 {...}
        |       REAL                                    {...}
        |       STRING                                  {...}
        ;
scanSwtch:                                              {...}
        |       COMMA INTVALUE COLON nxtType            {...}
        |       COMMA INTVALUE COLON nxtType SETFLEX    {...}
        |       COMMA IDENT COLON nxtType               {...}
        |       COMMA IDENT COLON nxtType SETFLEX       {...}
        ;
nxtType :       INTEGER                                 {...}
        |       REAL                                    {...}
        |       STRING                                  {...}
        |       RLITERAL                                {...}
        ;
ssco    :       aSet                                    {...}
        |       aSeq                                    {...}
        |       LAB choice RAB                          {...}
        |       LSQB type RSQB                          {...}
        |       LSQB typeList RSQB                      {...}
```

```
        |       LAB2 litChoice RAB2                          {...}
        ;

/*
 ************************************************************
 */
litSet  :       LBRACE RBRACE                                {...}
        |       LBRACE litList RBRACE                        {...}
        |       LBRACE litList sepClause RBASTERISK          {...}
        |       LBRACE litList sepClause RBPLUS              {...}
        ;
aSet    :       LBRACE type RBRACE                           {...}
        |       LBRACE type sepClause RBN number scanSwtch   {...}
        |       LBRACE typeList RBRACE                       {...}
        |       LBRACE type sepClause RBASTERISK             {...}
        |       LBRACE type sepClause RBPLUS                 {...}
        ;
/*
 ************************************************************
 */
litSeq  :       LCB RCB                                      {...}
        |       LCB litList RCB                              {...}
        |       LCB litList sepClause RCASTERISK             {...}
        |       LCB litList sepClause RCPLUS                 {...}
        ;
aSeq    :       LCB type RCB                                 {...}
        |       LCB type sepClause RCN number scanSwtch      {...}
        |       LCB typeList RCB                             {...}
        |       LCB type sepClause RCASTERISK                {...}
        |       LCB type sepClause RCPLUS                    {...}
        ;
sepClause:                                                   {...}
        |       SEPBARS literal                              {...}
        ;
number  :       exp                                          {...}
        ;

exp     :       INTVALUE                                     {...}
        |       IDENT                                        {...}
```

```
        |       exp PLUS exp                                    {...}
        |       exp MINUS exp                                   {...}
        |       exp ANYTKN exp                                  {...}
        |       exp DIVI exp                                    {...}
        |       LCB exp RCB                                     {...}
        ;
/*
 ****************************************************************
 */
litChoice:      litList                                         {...}
        |       litChoice BAR litList                           {...}
        ;
choice  :       type                                            {...}
        |       choice BAR type                                 {...}
        ;
%%
#include "a2bFlex.c"
/*
 ****************************************************************
 */
a2bParse(char *fn, struct ISPEC **iSpecp)
{
      int tmp, result = 0;
      char *t;
      FILE *fd;
      struct ISPEC *ifc;

      yyin = ( fn ? fopen(fn, "r") : fdopen(0, "r"));
      yy_init = 1;
      yy_start = 0;
      newRules = NULL;
      tRefers = NULL;
      litTable = NULL;
      fxdFields = NULL;
      lineNo = 1; casProd = 1; casOpt = 1;
      casRule = 0; casType = 0;

      result = parse();
```

```
        (*iSpecp)->fmod = interface->fmod;
        (*iSpecp)->lmod = interface->lmod;
        (*iSpecp)->next = interface->next;
        for (ifc = interface->next; ifc; ifc = ifc->next) {
              dupstr((*iSpecp)->spec, &(ifc->spec));
              dupstr((*iSpecp)->skelDir, &(ifc->skelDir));
        }
#ifdef LOG
        dumpCas2Log(cas);
#endif
        if (fn) {
              fclose(yyin); yyin = NULL;
        }

        return result;
}
/*
 ************************************************************
 */
static int yyerror(s) char *s;
{
        fprintf(stderr,
                "parse error near line %d with string %s\n", lineNo, yytext);
}
/*
 ************************************************************
 */
```

D.1.2 flex definition

```
%{
#undef YY_DECL
#define YY_DECL \
   static int a2bLex(yylvalp,yyllocp) YYSTYPE *yylvalp; YYLTYPE *yyllocp;
static char *strsave(char *s);
%}
D               [0-9]
E               [E][-+]{D}{D}
W               [\t ]
```

```
CHAR            [A-Za-z]
ALPHA           [!#-'\*-\-/\?-~]
ALPHANUM        [!#-'\*-\+\-/-9\?-~]
WHITE           [\t ]
%x code
%%
\+                                  { return PLUS; }
\-                                  { return MINUS; }
\/                                  { return DIVI; }
\#                                  { return CNTDECL; }
\*                                  { return ANYTKN; }
\@space                             { return DEFWSP; }
\@dfspace                           { return DFWSP; }
\%gofer\{\n?                        { BEGIN(code); return GOFERENV; }
\%mira\{\n?                         { BEGIN(code); return MIRAENV; }
\%c\{\n?                            { BEGIN(code); return CENV; }
<code>[^%\n]*                       { yymore(); }
<code>"%"+[^%}\n]*                  { yymore(); }
<code>\n                            { lineNo++; yymore(); }
<code>"%}"                          { int c = *(yytext+yyleng-2);
                                      *(yytext+yyleng-2) = '\0';
                                      (*yylvalp).string= strsave(yytext);
                                      *(yytext+yyleng-2) = c;
                                      BEGIN(INITIAL); return TRNSCODE;}
\%a2b                               { return A2B; }
\%({CHAR}|\_)({CHAR}|{D}|\-|\_)*    { (*yylvalp).string = strsave(yytext+1);
                                      return MODULENAME; }
\,                                  { return COMMA; }
\;                                  { return SEMI; }
\:                                  { return COLON; }
\|                                  { return BAR; }
\|\|                                { return SEPBARS; }
\{                                  { return LBRACE; }
\}                                  { return RBRACE; }
\}{W}*\+                            { return RBPLUS; }
\}{W}*\*                            { return RBASTERISK; }
\}{W}*\^                            { return RBN; }
\(                                  { return LCB; }
\)                                  { return RCB; }
```

```
\){W}*\+                                { return RCPLUS; }
\){W}*\*                                { return RCASTERISK; }
\){W}*\^                                { return RCN; }
\[                                      { return LSQB; }
\]                                      { return RSQB; }
\<                                      { return LAB; }
\>                                      { return RAB; }
\<\<                                    { return LAB2; }
\>\>                                    { return RAB2; }
\={W}*\(                                { return SREP; }
to                                      { return TO; }
via                                     { return VIA; }
{D}+                                    { (*yylvalp).intValue = atoi(yytext);
                                          return INTVALUE; }

setflexmode                             { return SETFLEX; }
literal                                 { (*yylvalp).string = strsave(yytext);
                                          return RLITERAL; }

integer                                 { (*yylvalp).string = strsave(yytext);
                                          return INTEGER; }

real                                    { (*yylvalp).string = strsave(yytext);
                                          return REAL; }

string                                  { (*yylvalp).string = strsave(yytext);
                                          return STRING; }

({CHAR})({CHAR}|{D}|\-)*                { (*yylvalp).string = strsave(yytext);
                                          return IDENT; }

\'([^\'\\]|(\\.))*\'                    { (*yylvalp).string = strsave(yytext);
                                          return WSPACE; }

\"([^\"\\]|(\\.))*\"                    { (*yylvalp).string = strsave(yytext);
                                          return LITERAL; }

[\n]                                    { lineNo++; }
{WHITE}*                                {}
%%
static char *strsave(s) char *s;
{
        char *t;
        t = (char *)malloc(strlen(s));
        strcpy(t, s);
        return t;
}
```

D.2 Differences

Generating a decoder module that decodes values represented using a fixed number of characters within a text-file is awkward and was not considered during the initial design and construction of a2b. Although notation for defining this kind of data is described earlier in Chapter 7.2.2.1, to simplify the implementation of a2b the same information has to be specified using a scan switch at particular points within the representation grammar.

More specifically, a *scan switch* defines the type of the next data value to be decoded and the number of characters used to represent this value. A scan switch must be given in a representation definition before any use of a simple type with a fixed-sized representation. Using BNF, a scan switch is defined as follows:

⟨scan switch⟩ ⟶ , ⟨size⟩ : ⟨simple type keyword⟩

⟨size⟩ ⟶ ⟨integer value⟩ | ⟨integer type name⟩

⟨simple type keyword⟩ ⟶ `integer`

| `real`

| `string`

To illustrate, consider a representation definition of one sequence data type that comprises two integer types. A value of the first is represented using two characters and indicates the number of characters used to represent the second. The definition of this representation is:

```
%swEg, 2:integer
swEg : "A value:" ( #integer1, integer1:integer integer2) ;
integer1: integer:2;
integer2 : integer:integer1;
```

A data value conforming to this representation definition is:

```
A value:041098
```

where the first integer value is 04 and the second is 1098. Notice in the above representation definition that a scan switch can follow the name of the representation definition to define the size and type of the first data value in the text-file.

The information within a scan switch is also specified by the definition of the type and size of the next integer, real, or string value to be decoded. Therefore, instead of using a scan switch a2b should use the information provided by the definition of the next simple type to be decoded. Determining the next simple type is not implemented within a2b at present. However, using techniques analogous to calculating lookahead sets for LL(k) grammars (Sudkamp 1988, Hopcroft & Ullman 1979), the author is confident that determining the next simple type could be accomplished in a future implementation of a2b.

Practical examples: transfer specifications

Appendix E

E.1 Transfer specification for the interface sdts

```
%a2b sdts to tmp

%sdts, 5:integer
sdts  : (ddr (dr)* );
ddr : (ddrLeader ddrDirectory "\036" (field1 "\036")* "\n", 5:integer);
dr : (drLeader drDirectory "\036" (field2 "\036")* "\n", 5:integer);
ddrLeader : (                                recordLength,
              1:integer                      interchangeLevel,
              1:string                       leaderIdentifier,
              1:string                       extensionIndicator,
              1:string                       reserved1,
              1:string                       applicationIndicator,
              2:integer                      #fieldControlLength,
              5:integer                      #addrsdda,
              3:string                       [extdCharSetIndicator,
              1:integer]                     #szFieldLength,
              1:integer                      #szFieldPosition,
              1:string                       reserved2,
              1:integer                      #szFieldTag,
                szFieldTag:string);
ddrDirectory: (directoryEntry)^(addrsdda - 24) /
```

```
                              (szFieldLength + szFieldPosition + szFieldTag);

directoryEntry: (                              fieldTag,
                szFieldLength:integer          fieldLength,
                szFieldPosition:integer        fieldPosition,
                szFieldTag:string);
drLeader : (                                   drLength,
                1:string                       reserved1,
                1:string                       drlIdentifier,
                5:string                       [reserved3,
                5:integer]                     #baseAddress,
                3:string                       [reserved4,
                1:integer]                     #szFieldLength,
                1:integer                      #szFieldPosition,
                1:string                       reserved2,
                1:integer                      #szFieldTag,
                szFieldTag:string);
drDirectory: (directoryEntry)^(baseAddress - 24)
                         / (szFieldLength + szFieldPosition + szFieldTag);
field1 : < (fileControlField name  [ "\037" tagPairs ])
         | (fieldStruc [name] "\037" [label] "\037" [fctrl])
         >;
field2 : (unit || "\037")*;
unit : ([data]);
fieldStruc : < elementaryField | bitField | collectionField > ;
fileControlField : string = ("0000;&":"%s");
elementaryField : string = ("0[1-4]00;&":"%s");
bitField : string = ("0500;&":"%s");
collectionField : string = ("[1-2][0-6]00;&":"%s");

tagPairs,name,label,fctrl,
data,data1,data2: string=("[^\037\036\;\&\n]+":"%s");

recordLength, addrsdda:                              integer:5;
interchangeLevel,szFieldPosition,
szFieldLength,szFieldTag:                            integer:1;
leaderIdentifier,reserved1,reserved2,drlIdentifier,
applicationIndicator,extensionIndicator:             string:1;
reserved4,extdCharSetIndicator:                      string:3;
```

```
fieldControlLength:                                    integer:2;
fieldLength:                                           integer:szFieldLength;
fieldPosition:                                         integer:szFieldPosition;
fieldTag:                                              string:szFieldTag;
baseAddress,drLength: integer:5;
reserved3 : string:5;

%tmp
tmp : (ddr (dr)* );
ddr : ("DDR Leader:\n" ddrLeader "\n"
       "DDR Directory:\n" ddrDirectory "\n"
       "DDR Data descriptive area:\n" (field1)*);
dr : ( "DR Leader:\n" drLeader "\n"
       "DR Directory:\n"  drDirectory  "\n"
       "DR Data descriptive area:\n" (field2 "\n")*);
ddrLeader : ( recordLength interchangeLevel leaderIdentifier
              extensionIndicator reserved1 applicationIndicator
              fieldControlLength addrsdda [extdCharSetIndicator]
              szFieldLength szFieldPosition reserved2 szFieldTag);
ddrDirectory: (directoryEntry||"\n")*;
directoryEntry:(fieldTag fieldLength fieldPosition);
drLeader : (drLength reserved1 drlIdentifier [reserved3] baseAddress [reserved4]
            szFieldLength szFieldPosition reserved2 szFieldTag);
drDirectory: (directoryEntry || "\n")*;
field1 : < (fileControlField "\n" "name:" name  "\n" [ "tag pairs:" tagPairs "\n" ])
         | (fieldStruc ["name:" name "\n" ]
                       ["Label:" label "\n"] ["Fctrl:" fctrl "\n" ])
         >;
field2 : (unit || ", " )*;
unit : ([data]);
fieldStruc : < elementaryField | bitField | collectionField > "\n";

fileControlField,elementaryField,bitField,collectionField,
tagPairs,name,label,fctrl,data,data1,data2: string;

fieldControlLength,fieldLength,fieldPosition,recordLength,
addrsdda,interchangeLevel,szFieldPosition,
szFieldLength,szFieldTag,baseAddress,drLength:              integer;
leaderIdentifier,reserved1,reserved2,reserved3,
```

```
reserved4,applicationIndicator,extensionIndicator,
extdCharSetIndicator,fieldTag,drlIdentifier:                string;

%sdts2tmp %c{
sdts2tmp(s,dp) struct type_SDTS_Sdts *s; struct type_TMP_Tmp **dp;
{
    *dp = (struct type_TMP_Tmp *) s;
}
%}
```

E.2 Transfer specification for the interface gfc

```
%a2b gia to coors
%gia
gia : "udb-feature" '\n' { Feature }* ;
Feature : ("feat" description
        [ ("coor" (point)* '\n')* ]
        [ "text" str '\n' ]
        [ "attr" (( attrValue || ",")+ || ",\natcr")+ '\n' ]);
description: ( id featNonSpatialType layerId networkId featSpatialType
         featGeomSpace gp1 gp2 flow '\n');
attrValue : ([< in | rl | str >]) ;
point : (x y);
featNonSpatialType : << "parcel" | "parcel_bdy" | "r_r_bdy" | "vinculum"
                      | "attr_node" | "r_r_hyd" | "road" | "r_r_arc"
                      | "landmark" | "pte_road" | "pte_rd_bdy"  | "pte_rd_arc"
                      | "frm_road" | "parcel_arc" | "railway" | "parcel_hyd" >>;
featSpatialType: << "l" | "n" | "p" | "pt" >>;
featGeomSpace: < << "xy" | "xyz">> | "xycz" constz > ;
str : string=("\"((\\.)|[^\"\\])*\"" : "\"%s\"");
x,y :  number ;
number : <i | r>;
r, rl, constz,gp1,gp2 : real;
i, in, id,layerId,networkId,flow : integer ;

%coors
coors : (crds)*;
crds:   (id seq x y '\n');
x,y :  real;
```

```
seq, id : integer ;

%gia2coors
%c{
#include "../define.h"

gia2coors(giaData,coorsp)
        struct type_GIA_Gia *giaData;
        struct type_COORS_Coors **coorsp;
{
      struct type_COORS_Coors *coors;
      struct type_GIA_CasS2 *crds;
      struct type_GIA_CasS0 *pnts;
      struct type_GIA_ParcelFeature *pf;
      double num();
      int seq;

      printf("performing gia2coors translation...\n");

      coors = NULL;
      for(; giaData; giaData=giaData->next) {
            seq = 0;
            for (crds = giaData->member_GIA_0->casS2; crds; crds=crds->next)
                  for (pnts=crds->element_GIA_1; pnts; pnts=pnts->next) {
                        nextValue(coors,coorsp,type_COORS_Coors);
                        coors->element_COORS_0 = new(type_COORS_Crds);
                        coors->element_COORS_0->id = giaData->member_GIA_0->elm1->id;
                        coors->element_COORS_0->seq = seq++;
                        coors->element_COORS_0->x = num(pnts->element_GIA_0->x);
                        coors->element_COORS_0->y = num(pnts->element_GIA_0->y);
            }
      }
}

double num(n) struct type_GIA_Number *n;
{
      switch(n->offset) {
         case type_GIA_Number_i : return (double) n->un.i;
         case type_GIA_Number_r : return n->un.r;
```

```
        default : ;
      }
}

%}
```

E.3 Transfer specification for the interface pcent

```
%a2b gia to parcels
%gia
gia : "udb-feature" '\n' { Feature }* ;
Feature : ("feat" id defn);

defn : < "parcel" parcelFeature | otherFeatures >;

parcelFeature : ( layerId networkId featSpatialType
                  featGeomSpace gp1 gp2 flow '\n'
    "coor"    x  y '\n'
  [ "text" str '\n' ]
    "attr"    [ a1 ] "," [ a2 ] "," [ a3 ] "," [ a4 ] "," [ a5 ] "," [ a6 ] ","
              [ a7 ] "," [ a8 ] "," [ a9 ] "," [ a10 ] "," [ a11 ] "," [ a12 ] ","
              [ a13 ] "," [ a14 ] "," [ a15 ] "," [ a16 ] "," [ a17 ] "," [ a18 ] ","
              [ a19 ]
    ",\natcr" [ a20 ] "," [ a21 ] "," [ a22 ] "," [ a23 ] "," [ a24 ] ","
              [ a25 ] "," [ a26 ] "," [ a27 ] "," [ a28 ] "," [ a29 ] ","
              [ a30 ] "," [ a31 ] "," [ a32 ] "," [ a33 ] "," [ a34 ] ","
              [ houseNumber ] "," [ streetAddress ] "," [ a37 ] );
otherFeatures : (
         featNonSpatialType layerId networkId featSpatialType
         featGeomSpace gp1 gp2 flow '\n'
        [ ("coor" (point)+ '\n')+ ]
        [ "text" str '\n' ]
        [ "attr" (( attrValue || ",")+ || ",\natcr")+ '\n' ]);
a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13,a14,a15,a16,a17,a18,a19,a20,
a21,a22,a23,a24,a25,a26,a27,a28,a29,a30,a31,a32,a33,a34,a37,
houseNumber, streetAddress, attrValue : < in | rl | str > ;
point : (x y);
featNonSpatialType : << "parcel_bdy" | "r_r_bdy" | "vinculum"
                     | "attr_node" | "r_r_hyd" | "road" | "r_r_arc"
```

```
                    | "landmark" | "pte_road" | "pte_rd_bdy"  | "pte_rd_arc"
                    | "frm_road" | "parcel_arc" | "railway" | "parcel_hyd" >>;
featSpatialType: << "l" | "n" | "p" | "pt" >>;
featGeomSpace: < << "xy" | "xyz">> | "xycz" constz > ;
str : string=("\"((\\.)|[^\"\\])*\"" : "\"%s\"");
x,y :  number ;
number : <i | r>;
r, rl, constz,gp1,gp2 : real;
i, in, id,layerId,networkId,flow : integer ;

%parcels
parcels : { Feature }* ;
Feature : ( id ":" x ":" y ":"
           [ houseNumber ] ":" [ houseLetter ] ":" [ streetAddress ]'\n') ;
x,y :  real;
id,houseNumber : integer ;
houseLetter,streetAddress : string=("[A-Z*][^:\n]*" : "%s");

%gia2parcels
%c{
#include "../define.h"
int atoi();

gia2parcels(giaData,parcelsDatap)
        struct type_GIA_Gia *giaData;
        struct type_PARCELS_Parcels **parcelsDatap;
{
      struct type_PARCELS_Parcels *pd;
      struct type_GIA_ParcelFeature *pf;
      double num();
      char *s0, *s1;
      printf("performing gia2parcels translation...\n");

      pf = NULL; *parcelsDatap = NULL; pd = NULL;
      while(giaData)  {
            if (giaData->member_GIA_0->elm1->offset == type_GIA_Defn_parcelFeature) {
                  nextValue(pd,parcelsDatap,type_PARCELS_Parcels);
                  pd->member_PARCELS_0 = new(type_PARCELS_Feature);
                  pf = giaData->member_GIA_0->elm1->un.parcelFeature;
```

```
                pd->member_PARCELS_0->id = giaData->member_GIA_0->id;
                pd->member_PARCELS_0->x = num(pf->x);
                pd->member_PARCELS_0->y = num(pf->y);
                if (pf->anon38->un.str) {
                    pd->member_PARCELS_0->optionals
                       = pd->member_PARCELS_0->optionals
                       | opt_PARCELS_Feature_houseNumber;
                    s0 = (char *)qb2str(pf->anon38->un.str);
                    for (s1=s0;isdigit(*s1);s1++) ;
                    pd->member_PARCELS_0->houseLetter = str2qb(s1,strlen(s1),0);
                    *s1 = '\0';
                    pd->member_PARCELS_0->houseNumber = atoi(s0);
                }
                pd->member_PARCELS_0->streetAddress = pf->anon39->un.str;
            }
            giaData = giaData->next;
        }
}
double num(n) struct type_GIA_Number *n;
{
        switch(n->offset) {
           case type_GIA_Number_i : return (double) n->un.i;
           case type_GIA_Number_r : return n->un.r;
           default : ;
        }
}
%}
```

E.4 Transfer specification for the interface cbdyxy

```
%a2b cbdyxy to cbdys

%cbdyxy
cbdyxy : (idxy)*;
idxy : (cbdy ":" x ":" y);
cbdy : integer;
x,y : real;

%cbdys
```

```
cbdys : (cbdy)*;
cbdy : ("boundary :" id '\n' points '\n');
points : (point)*;
point : ("(" x ", " y ")" '\n');
id : integer;
x,y : real;

%cbdyxy2cbdys %c{

#include "../define.h"
cbdyxy2cbdys(s,dp)
     struct type_CBDYXY_Cbdyxy *s;
     struct type_CBDYS_Cbdys **dp;
{
      struct type_CBDYS_Cbdys *d;
      struct type_CBDYS_Points *pnt;
      int id;

      id = -1;
      for (;s;s=s->next) {
            if (id != s->element_CBDYXY_0->cbdy) {
                  nextValue(d,dp,type_CBDYS_Cbdys);
                  d->element_CBDYS_0 = new(type_CBDYS_Cbdy);
                  d->element_CBDYS_0->id = s->element_CBDYXY_0->cbdy;
                  d->element_CBDYS_0->elm2 = NULL;
                  pnt=NULL;
                  id = d->element_CBDYS_0->id;
                  continue; /* skip first point, since it will be the last */
            }
            if (pnt) {
                  if ((pnt->element_CBDYS_1->x == s->element_CBDYXY_0->x)
                        && (pnt->element_CBDYS_1->y == s->element_CBDYXY_0->y))
                        continue; /* skip point */
            }
            nextValue(pnt,&(d->element_CBDYS_0->elm2),type_CBDYS_Points);
            pnt->element_CBDYS_1 = new(type_CBDYS_Point);
            pnt->element_CBDYS_1->x = s->element_CBDYXY_0->x;
            pnt->element_CBDYS_1->y = s->element_CBDYXY_0->y;
      }
```

```
}%}
```

E.5 Transfer specification for the interface pip

```
%a2b in to featcbdys

%in
in : ( parcels cbdys);
parcels : { Feature }* ;
Feature : ( id ":" x ":" y ":"
            [ houseNumber ] ":" [ houseLetter ] ":" [ streetAddress ] '\n') ;
houseNumber: integer ;
houseLetter,streetAddress : string=("[A-Z*][^:\n]*" : "%s");
cbdys : (cbdy)*;
cbdy : ("boundary :" id '\n' points '\n');
points : (point)*;
point : ("(" x ", " y ")" '\n');
id : integer;
x,y : real;

%featcbdys
featcbdys : (featcbdy)*;
featcbdy : (pid cbdy '\n');
pid,cbdy : integer;

%in2featcbdys %c{
#include "../define.h"
#include <math.h>

struct PointList {
      struct Point {
            double x,y;
      } *point;
      struct PointList *next;
}

in2featcbdys(in,fcp)
     struct type_IN_In *in;
     struct type_FEATCBDYS_Featcbdys **fcp;
```

```
{
        struct type_IN_Parcels *parcel;
        struct type_IN_Cbdys *cbdy;
        struct type_FEATCBDYS_Featcbdys *fc;
        struct Point pnt;

        fc = NULL;
        for (parcel = in->elm0; parcel; parcel = parcel->next) {
                pnt.x = parcel->member_IN_0->x;
                pnt.y = parcel->member_IN_0->y;
                for (cbdy = in->elm1; cbdy; cbdy = cbdy->next)
                        if (pointInpolygon(
                                &pnt,(struct PointList *)cbdy->element_IN_0->elm10))
                        {
                                nextValue(fc,fcp,type_FEATCBDYS_Featcbdys);
                                fc->element_FEATCBDYS_0 = new(type_FEATCBDYS_Featcbdy);
                                fc->element_FEATCBDYS_0->pid = parcel->member_IN_0->id;
                                fc->element_FEATCBDYS_0->cbdy = cbdy->element_IN_0->id;
                                break; /* find only the first */
                        }
        }
}
int pointInpolygon(pnt,ply) struct Point *pnt; struct PointList *ply;
{
        struct PointList *p, *np;
        int r;
        double maxx, miny, diffx, diffy;

        for (p = ply; p; p=p->next) {
                diffx = fabs(pnt->x - p->point->x); diffy = abs(pnt->y - p->point->y);
                maxx = (p == ply || (diffx > maxx) ? diffx : maxx);
                if (diffy != 0.0)
                        miny = (p == ply || (miny > diffy) ? diffy : miny);
        }
        for (p = ply, r=0; p; p=p->next) {
                np = ( p->next ? p->next : ply);
                r = r + lintersect(pnt->x, pnt->y, pnt->x + 2.0 * maxx, pnt->y + miny,
                                p->point->x, p->point->y, np->point->x, np->point->y);
        }
```

```
      return (r % 2);
}

#define det(a,b,c,d) ((a) * (d)) - ((c) * (b))

int lintersect(x0,y0,x1,y1,xp0,yp0,xp1,yp1)
     double x0,y0,x1,y1,xp0,yp0,xp1,yp1;
{
      double d,d1,d2;
      int r;

      d  = det( x1 - x0, xp0 - xp1,  y1 - y0, yp0 - yp1);
      d1 = det(xp0 - x0, xp0 - xp1, yp0 - y0, yp0 - yp1);
      d2 = det( x1 - x0, xp0 - x0,   y1 - y0, yp0 - y0);
      r = ((d1 * (d - d1)) >= 0.0) && ((d2 * (d - d2)) >= 0.0);
      return r;
}

%}
```

Templates

Appendix F

This Appendix contains the templates used by a2b as the basis for constructing the various modules forming an interface. Within each template there are lines prefixed by the symbol @%. The software tool a2b replaces these lines with code that is unique to the interface module being generated.

F.1 Non-communicating interface

```
/*
 *        main program for an Interface generated by a2b; don't edit.
 *
 *        interface.skel version 0
 */
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/param.h>
#include <string.h>
#include <varargs.h>

@% put -types.h include and *..Data = NULL decls for each representation definition

static char *datafile = NULL, *outFile = NULL, *fullName();

extern char *optarg;
extern int optind;

static int errflg = 1;
static char *tmpPath, *tmpFile();

main(argc, argv) int argc; char **argv;
{
      FILE *fd, *sfd, *dfd;
      int c;
      char *s, t;

      while ((c = getopt(argc, argv, "o:")) != -1)
            switch (c) {
               case 'o':
```

```
                errflg--;
                outFile = fullName(optarg);
                fd = fopen(outFile, "w");
                if (!fd) {
                      fprintf(stderr,
                              "Unable to open destination file %s for writting\n", outFile);
                      _exit(5);
                }
                fclose(fd); free(fd); fd = NULL;
                break;
             case '?':
                errflg++;
                break;
          }
      datafile = fullName(argv[optind]);
      if (datafile == NULL || errflg) {
@% fprintf(stderr, "usage: interface -o outFileName dataFileName\n");
          _exit(1);
      }
      tmpPath = (char *)malloc(strlen(datafile));
      strcpy(tmpPath, datafile);
      for (s = tmpPath+strlen(tmpPath); *s!= '/'; s--) ;
      *(s+1) = '\0';

@% interface body
      return 0;
}
/*
 ****************************************************************
 *
 *   shell(format, arg...)
 *
 *   Formats the args a la printf and feeds the
 *   result to system(). Exits if there's an error.
 *
 */
shell(va_alist) va_dcl
{
      va_list ap;
      char *format;
      char buf[4*MAXPATHLEN];
      int status;

      va_start(ap);
      format = va_arg(ap, char *);
      vsprintf(buf, format, ap);
      status = system(buf) >> 8;
      if (status) exit(status);
}
/*
 ****************************************************************
 */
static char *fullName(f) char *f;
{
        char *t, *s, tmp[MAXPATHLEN];

        if (f == NULL) return NULL;
        realpath(f, tmp);
        t = strstr(tmp, "/users/");
        s = (char *)malloc(strlen(t)+1);
        strcpy(s, t);

        return s;
```

```
}
/*
 ****************************************************************
 */
static char *tmpFile(tmpPath, s) char *tmpPath, *s;
{
      char *n;

      n = (char *)malloc(strlen(s)+strlen(tmpPath)+1);
      sprintf(n, "%s%s", tmpPath, s);

      return n;
}
/*
 ****************************************************************
 */
```

F.2 Communicating interface

F.2.1 Initiator

```
#include <ctype.h>
#include <stdio.h>
#include <unistd.h>
#include <pwd.h>
#include <varargs.h>
#ifndef PEPSY_VERSION
#define PEPSY_VERSION 1
#endif
#include "rosy.h"

@% put #include and *..Data = NULL decls for each representation definition
/*
 ********************************************************************************
 */

struct dispatch {
      char   *ds_name;
      int            ds_operation;

      IFP            ds_argument;
      modtyp *ds_fr_mod;        /* pointer to table for arguement type */
      int     ds_fr_index;      /* index to entry in tables */

      IFP            ds_result;
      IFP            ds_error;

      char   *ds_help;
};

/********************************************************************************
 *
 * ISODE communication service identification values
 */
static char
      *myname = "ryinitiator",
@% static char *myservice, *mycontext,*mypci = "....geodata transfer", ...

/********************************************************************************
 *
 * External functions
```

```
 */

extern int
      /*
       * system function for checking availability of a file
       */
      access();

extern void
      /*
       * ISODE Functions
       */
      adios(), advise(), acs_adios(), acs_advise(), ros_adios(), ros_advise();


/*
 ****************************************************************************
 */

static int
      ryinitiator (int argc, char  **argv, char *myservice, char *mycontext, char *mypci,
             struct RyOperation ops[], struct dispatch *dispatches, IFP quit);

/*****************************************************************************
 *
 * Functions for processing data before communication.
 */

static int
      do_getFile(int sd, struct dispatch *ds, char  **args,
                 struct type_A2BComModName_IA5List **ia5),

      do_help(int sd, struct dispatch *ds, char  **args,
              caddr_t *dummy),

      do_quit(int sd, struct dispatch *ds, char  **args,
              caddr_t *dummy);

/*****************************************************************************
 *
 * Function for processing data after communication.
 */

static int
      interfaceBody (int sd, int id, int dummy,
                     struct type_A2BComModName_A2BCmn *A2Bcmn,
                     struct RoSAPindication *roi);
/*****************************************************************************
 *
 * Function for processing errors.
 */

static int
      error (int sd, int id, int error,
             struct type_A2BComModName_IA5List *parameter,
             struct RoSAPindication *roi);
/*****************************************************************************
 *
 * Miscellaneous functions
 */

static int
      print_ia5list (struct type_A2BComModName_IA5List *ia5);
```

```
static struct type_A2BComModName_IA5List
      *vec2ia5list (char  **vec);

/*
 ************************************************************************
 */
static struct dispatch dispatches[] = {

    "getFile",        operation_A2BComModName_getFile,
    do_getFile, &_ZA2BComModName_mod, _ZIA5ListA2BComModName,
    interfaceBody, error,
    "SYNOPSIS\n\tgetFile srcFilename destFilename\n\nDESCRIPTION\n\
     \tTransfer the named source datafile to the named destination datafile\n",

    "help", 0,
    do_help, NULL, 0,
    NULLIFP, NULLIFP,
    "print this information",

    "quit", 0,
    do_quit, NULL, 0,
    NULLIFP, NULLIFP,
    "terminate the association and exit",

    NULL
};

static char *outFile = NULL;

/************************************************************************
 *
 * Main function
 */
main (int argc, char  **argv, char  **envp)
{
    ryinitiator (argc, argv, myservice, mycontext, mypci,
                 table_A2BComModName_Operations, dispatches, do_quit);

    exit (0);                       /* NOTREACHED */
}
/*
 ********************************************************
 */
static int do_getFile(int sd, struct dispatch *ds, char  **args,
                  struct type_A2BComModName_IA5List **ia5)
{
      int error;
      struct type_A2BComModName_IA5List *dn;

      error = 0;
      if ( !( *ia5 = vec2ia5list(args))) {
            advise(ds->ds_name, "Error setting source filename");
            error++;
      }
      dn = (*ia5)->next;
      if (dn)
            outFile = (char *) qb2str(dn->element_A2BComModName_1);
      else {
            advise(ds->ds_name, "Error setting destination filename");
            error++;
      }
```

```
    return (error ? NOTOK : OK);
}
/*
 ********************************************************
 */
static int  do_help(int sd, struct dispatch *ds, char  **args,
                    caddr_t *dummy)
{
    (void) printf ("\nCommands are:\n");
    for (ds = dispatches; ds -> ds_name; ds++)
        (void) printf ("%s\t%s\n", ds -> ds_name, ds -> ds_help);

    return NOTOK;
}
/*
 ********************************************************
 */
static int  do_quit(int sd, struct dispatch *ds, char  **args,
                    caddr_t *dummy)
{
    struct AcSAPrelease acrs;
    register struct AcSAPrelease   *acr = &acrs;
    struct AcSAPindication  acis;
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort *aca = &aci -> aci_abort;

    if (AcRelRequest (sd, ACF_NORMAL, NULLPEP, 0, NOTOK, acr, aci) == NOTOK)
        acs_adios (aca, "A-RELEASE.REQUEST");

    if (!acr -> acr_affirmative) {
        (void) AcUAbortRequest (sd, NULLPEP, 0, aci);
        adios (NULLCP, "release rejected by peer: %d", acr -> acr_reason);
    }

    ACRFREE (acr);

    exit (0);
}
/*
 ********************************************************
 */
static int interfaceBody (int sd, int id, int dummy,
                          struct type_A2BComModName_A2BCmn *A2Bcmn,
                          struct RoSAPindication *roi)
{
     FILE *fd;

@% the main routine of destination communicating interface

     return OK;
}
/*
 ********************************************************
 */
static int error (int sd, int id, int error,
                  struct type_A2BComModName_IA5List *parameter,
                  struct RoSAPindication *roi)
{
    register struct RyError *rye;

    if (error == RY_REJECT) {
        advise (NULLCP, "%s", RoErrString ((int) parameter));
        return OK;
```

```
    }

    if (rye = finderrbyerr (table_A2BComModName_Errors, error))
        advise (NULLCP, "%s",  rye -> rye_name);
    else
        advise (NULLCP, "Error %d", error);

    if (parameter)
        print_ia5list (parameter);

    return OK;
}
/*
 ***************************** TYPES ****************************
 */
static struct type_A2BComModName_IA5List *vec2ia5list (char  **vec)
{
    struct type_A2BComModName_IA5List  *ia5;
    register struct type_A2BComModName_IA5List **ia5p;

    ia5 = NULL;
    ia5p = &ia5;

    for (; *vec; vec++) {
        if ((*ia5p = (struct type_A2BComModName_IA5List *) calloc (1, sizeof **ia5p))
                == NULL)
            adios (NULLCP, "out of memory");

        if (((*ia5p) -> element_A2BComModName_1 = str2qb (*vec, strlen (*vec), 1)) == NULL)
            adios (NULLCP, "out of memory");

        ia5p = &((*ia5p) -> next);
    }

    return ia5;
}

static int print_ia5list (struct type_A2BComModName_IA5List *ia5)
{
    register struct qbuf *p,
                         *q;

    for (; ia5; ia5 = ia5 -> next) {
        p = ia5 -> element_A2BComModName_1;
        for (q = p -> qb_forw; q != p ; q = q -> qb_forw)
            (void) printf ("%*.*s", q -> qb_len, q -> qb_len, q -> qb_data);
        (void) printf ("\n");
    }
}
/*
 ***************************************************************************
 */

/* DATA */

static int count = 1;
int        length = 536;

#define        DS_RESULT(ds)        ((ds) -> ds_result)

static int  getline ();
static void invoke ();
```

```
extern char *isodeversion;

static int ryinitiator (int argc, char  **argv,
                        char *myservice, char *mycontext, char *mypci,
                        struct RyOperation ops[], struct dispatch *dispatches, IFP quit)
{
    int             iloop,
            sd;
    register char  *cp,
                  **ap;
    char    buffer[BUFSIZ],
           *vec[NVEC + 1];
    register struct dispatch   *ds;
    struct QOStype qos;
    struct SSAPref sfs;
    register struct SSAPref *sf;
    register struct PSAPaddr *pa;
    struct AcSAPconnect accs;
    register struct AcSAPconnect   *acc = &accs;
    struct AcSAPindication  acis;
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort *aca = &aci -> aci_abort;
    AEI             aei;
    OID             ctx,
            pci;
    struct PSAPctxlist pcs;
    register struct PSAPctxlist *pc = &pcs;
    struct RoSAPindication rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject *rop = &roi -> roi_preject;

    if (myname = rindex (argv[0], '/'))
        myname++;
    if (myname == NULL || *myname == NULL)
        myname = argv[0];

    isodetailor (myname, 1);

    qos.qos_reliability = HIGH_QUALITY;

    for (ap = argv + 1; cp = *ap; ap++) {
        if (*cp != '-')
            break;

        if (strcmp (cp, "-low") == 0) {
            qos.qos_reliability = LOW_QUALITY;
            continue;
        }
        if (strcmp (cp, "-high") == 0) {
            qos.qos_reliability = HIGH_QUALITY;
            continue;
        }
        if (strcmp (cp, "-c") == 0) {
            if ((cp = *++ap) == NULL
                    || sscanf (cp, "%d", &count) != 1
                    || count < 1)
                adios (NULLCP, "usage: %s -c count", myname);
            continue;
        }
        if (strcmp (cp, "-l") == 0) {
            if ((cp = *++ap) == NULL
                    || sscanf (cp, "%d", &length) != 1
                    || length < 0)
```

```
            adios (NULLCP, "usage: %s -l length", myname);
        continue;
    }

    adios (NULLCP, "%s: unknown switch", cp);
}

if ((cp = *ap++) == NULL)
    adios (NULLCP, "usage: %s host [operation [ arguments ... ]]", myname);

if ((aei = _str2aei (cp, myservice, mycontext, *ap == NULL, NULLCP,
                     NULLCP)) == NULLAEI)
    adios (NULLCP, "unable to resolve service: %s", PY_pepy);
if ((pa = aei2addr (aei)) == NULLPA)
    adios (NULLCP, "address translation failed");

if ((ctx = ode2oid (mycontext)) == NULLOID)
    adios (NULLCP, "%s: unknown object descriptor", mycontext);
if ((ctx = oid_cpy (ctx)) == NULLOID)
    adios (NULLCP, "out of memory");
if ((pci = ode2oid (mypci)) == NULLOID)
    adios (NULLCP, "%s: unknown object descriptor", mypci);
if ((pci = oid_cpy (pci)) == NULLOID)
    adios (NULLCP, "out of memory");
pc -> pc_nctx = 1;
pc -> pc_ctx[0].pc_id = 1;
pc -> pc_ctx[0].pc_asn = pci;
pc -> pc_ctx[0].pc_atn = NULLOID;

if ((sf = addr2ref (PLocalHostName ())) == NULL) {
    sf = &sfs;
    (void) bzero ((char *) sf, sizeof *sf);
}

if (*ap == NULL) {
    (void) printf ("%s", myname);
    if (sf -> sr_ulen > 2)
        (void) printf (" running on host %s", sf -> sr_udata + 2);
    if (sf -> sr_clen > 2)
        (void) printf (" at %s", sf -> sr_cdata + 2);
    (void) printf (" [%s, ", oid2ode (ctx));
    (void) printf ("%s]\n", oid2ode (pci));
    (void) printf ("using %s\n", isodeversion);

    (void) printf ("%s... ", cp);
    (void) fflush (stdout);

    iloop = 1;
}
else {
    cp = *ap++;
    for (ds = dispatches; ds -> ds_name; ds++)
        if (strcmp (ds -> ds_name, cp) == 0)
            break;
    if (ds -> ds_name == NULL)
        adios (NULLCP, "unknown operation \"%s\"", cp);

    iloop = 0;
}

if (AcAssocRequest (ctx, NULLAEI, aei, NULLPA, pa, pc, NULLOID,
            0, ROS_MYREQUIRE, SERIAL_NONE, 0, sf, NULLPEP, 0, &qos,
            acc, aci)
```

```
            == NOTOK)
        acs_adios (aca, "A-ASSOCIATE.REQUEST");

    if (acc -> acc_result != ACS_ACCEPT) {
        if (iloop)
            (void) printf ("failed\n");

        adios (NULLCP, "association rejected: [%s]",
                AcErrString (acc -> acc_result));
    }
    if (iloop) {
        (void) printf ("connected\n");
        (void) fflush (stdout);
    }

    sd = acc -> acc_sd;
    ACCFREE (acc);

    if (RoSetService (sd, RoPService, roi) == NOTOK)
        ros_adios (rop, "set RO/PS fails");

    if (iloop) {
        for (;;) {
            if (getline (buffer) == NOTOK)
                break;

            if (str2vec (buffer, vec) < 1)
                continue;

            for (ds = dispatches; ds -> ds_name; ds++)
                if (strcmp (ds -> ds_name, vec[0]) == 0)
                    break;
            if (ds -> ds_name == NULL) {
                advise (NULLCP, "unknown operation \"%s\"", vec[0]);
                continue;
            }

            invoke (sd, ops, ds, vec + 1);
        }
    }
    else
        invoke (sd, ops, ds, ap);

    (*quit) (sd, (struct dispatch *) NULL, (char **) NULL, (caddr_t *) NULL);
}

static void invoke (sd, ops, ds, args)
int        sd;
struct RyOperation ops[];
register struct dispatch *ds;
char  **args;
{
    register int    i;
    int            cc,
            result;
    caddr_t in;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    in = NULL;
    if (ds -> ds_argument && (*ds -> ds_argument) (sd, ds, args, &in) == NOTOK)
        return;
```

```
    for (i = 0; i < count; i++)
        switch (result = RyStub (sd, ops, ds -> ds_operation, RyGenID (sd),
                                 NULLIP, in, DS_RESULT (ds), ds -> ds_error,
                                 ROS_SYNC, roi)) {
            case NOTOK:                 /* failure */
                if (ROS_FATAL (rop -> rop_reason))
                    ros_adios (rop, "STUB");
                ros_advise (rop, "STUB");
                goto out;

            case OK:                /* got a result/error response */
                break;

            case DONE:                /* got RO-END? */
                adios (NULLCP, "got RO-END.INDICATION");
                /* NOTREACHED */

            default:
                adios (NULLCP, "unknown return from RyStub=%d", result);
                /* NOTREACHED */
        }

out: ;
    if (ds -> ds_fr_mod && in)
        (void) fre_obj (in, ds -> ds_fr_mod -> md_dtab[ds -> ds_fr_index],
                        ds -> ds_fr_mod, 1);
}

static int  getline (buffer)
char   *buffer;
{
    register int    i;
    register char  *cp,
                   *ep;
    static int  sticky = 0;

    if (sticky) {
        sticky = 0;
        return NOTOK;
    }

    (void) printf ("%s> ", myname);
    (void) fflush (stdout);

    for (ep = (cp = buffer) + BUFSIZ - 1; (i = getchar ()) != '\n';) {
        if (i == EOF) {
            (void) printf ("\n");
            clearerr (stdin);
            if (cp != buffer) {
                sticky++;
                break;
            }

            return NOTOK;
        }

        if (cp < ep)
            *cp++ = i;
    }
    *cp = NULL;

    return OK;
```

```
}

void        ros_adios (rop, event)
register struct RoSAPpreject *rop;
char   *event;
{
    ros_advise (rop, event);

    _exit (1);
}


void        ros_advise (rop, event)
register struct RoSAPpreject *rop;
char   *event;
{
    char    buffer[BUFSIZ];

    if (rop -> rop_cc > 0)
        (void) sprintf (buffer, "[%s] %*.*s", RoErrString (rop -> rop_reason),
                rop -> rop_cc, rop -> rop_cc, rop -> rop_data);
    else
        (void) sprintf (buffer, "[%s]", RoErrString (rop -> rop_reason));

    advise (NULLCP, "%s: %s", event, buffer);
}

void        acs_adios (aca, event)
register struct AcSAPabort *aca;
char   *event;
{
    acs_advise (aca, event);

    _exit (1);
}

void        acs_advise (aca, event)
register struct AcSAPabort *aca;
char   *event;
{
    char    buffer[BUFSIZ];

    if (aca -> aca_cc > 0)
        (void) sprintf (buffer, "[%s] %*.*s",
                AcErrString (aca -> aca_reason),
                aca -> aca_cc, aca -> aca_cc, aca -> aca_data);
    else
        (void) sprintf (buffer, "[%s]", AcErrString (aca -> aca_reason));

        advise (NULLCP, "%s: %s (source %d)", event, buffer,
                aca -> aca_source);
}

#ifndef        lint
static void        _advise ();

void        adios (va_alist)
va_dcl
{
    va_list ap;

    va_start (ap);
```

```
    _advise (ap);

    va_end (ap);

    _exit (1);
}
#else

void        adios (what, fmt)
char   *what,
       *fmt;
{
    adios (what, fmt);
}
#endif


#ifndef        lint
void        advise (va_alist)
va_dcl
{
    va_list ap;

    va_start (ap);

    _advise (ap);

    va_end (ap);
}

static void  _advise (ap)
va_list        ap;
{
    char    buffer[BUFSIZ];

    asprintf (buffer, ap);

    (void) fflush (stdout);

    (void) fprintf (stderr, "%s: ", myname);
    (void) fputs (buffer, stderr);
    (void) fputc ('\n', stderr);

    (void) fflush (stderr);
}
#else
void        advise (what, fmt)
char   *what,
       *fmt;
{
    advise (what, fmt);
}
#endif

#ifndef        lint
void        ryr_advise (va_alist)
va_dcl
{
    va_list ap;

    va_start (ap);

    _advise (ap);
```

```
    va_end (ap);
}
#else
void        ryr_advise (what, fmt)
char    *what,
        *fmt;
{
    ryr_advise (what, fmt);
}
#endif
```

F.2.2 Responder

```
#include <ctype.h>
#include <stdio.h>
#include <unistd.h>
#include <utmp.h>
#include <sys/stat.h>
#include <sys/param.h>
#include <varargs.h>
#ifndef PEPSY_VERSION
#define PEPSY_VERSION 1
#endif
#include "rosy.h"
#include "logger.h"

@% put -types.h include and *..Data = NULL decls for each representation definition

/*****************************************************************************
 *
 * ISODE communication service identification values
 */
static char
@% static char *myservice = ".... geodata transfer"

/*****************************************************************************
 *
 * External functions
 */

extern int
      /*
       * system function for checking availability of a file
       */
      access();
extern void
      /*
       * ISODE Functions
       */
      adios(), advise(), acs_advise(), ros_adios(), ros_advise(), ryr_advise ();

struct dispatch {
    char    *ds_name;
    int             ds_operation;

    IFP             ds_vector;
};

/*****************************************************************************
 *
```

```
 *      The 'function' sendData
 */

#define sendData(A) \
      if (RyDsResult(sd, rox->rox_id, A, ROS_NOPRIO, roi) == NOTOK)\
      ros_adios(&roi->roi_preject, "RESULT");
/*
 *
 ********************************************************************************/
static int execuid = 1, execgid = 1,

      ryresponder (int argc, char  **argv, char  *host, char  *myservice,
                   char *mycontext, struct dispatch *dispatches,
                   struct RyOperation *ops, IFP start, IFP stop),

      interfaceBody(int sd, struct RyOperation *ryo, struct RoSAPinvoke *rox,
                        caddr_t in, struct RoSAPindication *roi),

      ureject(int sd, int reason, struct RoSAPinvoke *rox,
                    struct RoSAPindication *roi),

      error (int sd, int err, caddr_t param,
             struct RoSAPinvoke *rox, struct RoSAPindication *roi);

static struct type_A2BComModName_IA5List *str2ia5list (char *s);

static struct dispatch dispatches[] = {
    "getFile", operation_A2BComModName_getFile, interfaceBody,

    NULL
};

/*
 ***************************** MAIN ******************************
 */

main (argc, argv, envp)
int        argc;
char  **argv,
      **envp;
{
    ryresponder (argc, argv, PLocalHostName (), myservice, NULLCP,
                 dispatches, table_A2BComModName_Operations, NULLIFP, NULLIFP);

    exit (0);                          /* NOTREACHED */
}

/*
 ***************************** OPERATIONS *****************************
 */

static int interfaceBody(int sd, struct RyOperation *ryo, struct RoSAPinvoke *rox,
                         caddr_t in, struct RoSAPindication *roi)
{
      int result;
      char *datafile, *tmpPath, *s, *tmpFile();
      FILE *fd;

      if ((result =doPreamble(sd, ryo, rox, (struct type_A2BComModName_IA5List  *) in,
                    roi, &datafile)) == NOTOK)
            goto out;

@% debugging loop - if requested
```

```
        tmpPath = (char *)malloc(strlen(datafile));
        strcpy(tmpPath, datafile);
        for (s = tmpPath+strlen(tmpPath); *s!= '/'; s--) ;
        *(s+1) = '\0';

@% the interface body generated by a2b, including if (RyDsResult(sd, ...) == NOTOK)...

        result = OK;

    out:;
        free(datafile);

        return result;

}
/*
 ***************************** doPreamble ***************************
 */
static int doPreamble(sd, ryo, rox, in, roi, datafilep)
     int        sd;
     struct RyOperation *ryo;
     struct RoSAPinvoke *rox;
     struct type_A2BComModName_IA5List  *in;
     struct RoSAPindication *roi;
     char **datafilep;
{
        int result;

        result = OK;
        if (rox -> rox_nolinked == 0) {
              advise (LLOG_EXCEPTIONS, NULLCP,
                      "RO-INVOKE.INDICATION/%d: %s, unknown linkage %d",
                      sd, ryo -> ryo_name, rox -> rox_linkid);
              result = ureject (sd, ROS_IP_LINKED, rox, roi);
        }
        else {
              advise (LLOG_NOTICE, NULLCP, "RO-INVOKE.INDICATION/%d: %s",
                      sd, ryo -> ryo_name);
              *datafilep =
                    (char *) qb2str (in->element_A2BComModName_1);
              if (access(*datafilep, R_OK) == NOTOK) {
                    result = error(sd, error_A2BComModName_unknownFile, NULLCP, rox, roi);
                    advise (LLOG_EXCEPTIONS, NULLCP, "unknown file %s", *datafilep);
                    free_A2BComModName_IA5List(in);
              }
        }
        return result;
}
/*
 ***************************** ERROR ***************************
 */
static int error (int sd, int err, caddr_t param,
                  struct RoSAPinvoke *rox, struct RoSAPindication *roi)
{
    if (RyDsError (sd, rox -> rox_id, err, param, ROS_NOPRIO, roi) == NOTOK)
        ros_adios (&roi -> roi_preject, "ERROR");

    return OK;
}

/*
 ***************************** U-REJECT ***************************
 */
```

```
static int ureject (int sd, int reason, struct RoSAPinvoke *rox,
                    struct RoSAPindication *roi)
{
    if (RyDsUReject (sd, rox -> rox_id, reason, ROS_NOPRIO, roi) == NOTOK)
        ros_adios (&roi -> roi_preject, "U-REJECT");

    return OK;
}
/*
 ***************************** TYPES ******************************
 */
static struct type_A2BComModName_IA5List *str2ia5list (char *s)
{
    register struct type_A2BComModName_IA5List *ia5;

    ia5 = (struct type_A2BComModName_IA5List  *)
          calloc(1, sizeof(struct type_A2BComModName_IA5List *));
    if (ia5 == NULL)
        return NULL;

    if ((ia5 -> element_A2BComModName_1 = str2qb (s, strlen (s), 1)) == NULL) {
        free ((char *) ia5);
        return NULL;
    }

    return ia5;
}
/*
 ****************************************************************
 */
static char *tmpFile(tmpPath, s) char *tmpPath, *s;
{
      char *n;

      n = (char *)malloc(strlen(s)+strlen(tmpPath)+1);
      sprintf(n, "%s%s", tmpPath, s);

      return n;
}
/*
 ****************************************************************
 *
 *   shell(format, arg...)
 *
 *   Formats the args a la printf and feeds the
 *   result to system(). Exits if there's an error.
 *
 */
shell(va_alist) va_dcl
{
      va_list ap;
      char *format;
      char buf[4*MAXPATHLEN];
      int status;

      va_start(ap);
      format = va_arg(ap, char *);
      vsprintf(buf, format, ap);
      status = system(buf) >> 8;
/*      if (status) exit(status); */
}
/*
```

```
 ****************************************************************
 */

/* generic idempotent responder (from imisc,ISODE) */

#include <stdio.h>
#include <setjmp.h>
#include <varargs.h>
#include "tsap.h"                /* for listening */
#include "tailor.h"

/*  DATA */

int        debug = 0;

static LLog _pgm_log = {
    "responder.log", NULLCP, NULLCP,
    LLOG_FATAL | LLOG_EXCEPTIONS | LLOG_NOTICE, LLOG_FATAL, -1,
    LLOGCLS | LLOGCRT | LLOGZER, NOTOK
};
LLog *pgm_log = &_pgm_log;

static char *myname = "ryresponder";

static jmp_buf toplevel;

static IFP        startfnx;
static IFP        stopfnx;

static int        ros_init (), ros_work (), ros_indication (), ros_lose ();

/*   RESPONDER */

static int ryresponder (int argc, char  **argv, char  *host, char  *myservice,
             char  *mycontext, struct dispatch *dispatches, struct RyOperation *ops,
             IFP start, IFP stop)
{
    register struct dispatch   *ds;
    AEI            aei;
    struct TSAPdisconnect   tds;
    struct TSAPdisconnect  *td = &tds;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (myname = rindex (argv[0], '/'))
        myname++;
    if (myname == NULL || *myname == NULL)
        myname = argv[0];

    isodetailor (myname, 0);
    if (debug = isatty (fileno (stderr)))
        ll_dbinit (pgm_log, myname);
    else {
        static char  myfile[BUFSIZ];

        (void) sprintf (myfile, "%s.log",
                        (strncmp (myname, "ros.", 4)
                                    && strncmp (myname, "lpp.", 4))
                                || myname[4] == NULL
                            ? myname : myname + 4);
        pgm_log -> ll_file = myfile;
        ll_hdinit (pgm_log, myname);
```

```
    }

    advise (LLOG_NOTICE, NULLCP, "starting");

    if ((aei = _str2aei (host, myservice, mycontext, 0, NULLCP, NULLCP))
            == NULLAEI)
        adios (NULLCP, "unable to resolve service: %s", PY_pepy);

    for (ds = dispatches; ds -> ds_name; ds++)
        if (RyDispatch (NOTOK, ops, ds -> ds_operation, ds -> ds_vector, roi)
                == NOTOK)
            ros_adios (rop, ds -> ds_name);

    startfnx = start;
    stopfnx = stop;

    if (isodeserver (argc, argv, aei, ros_init, ros_work, ros_lose, td)
            == NOTOK) {
        if (td -> td_cc > 0)
            adios (NULLCP, "isodeserver: [%s] %*.*s",
                    TErrString (td -> td_reason),
                    td -> td_cc, td -> td_cc, td -> td_data);
        else
            adios (NULLCP, "isodeserver: [%s]",
                    TErrString (td -> td_reason));
    }

    exit (0);
}

/*  */

static int  ros_init (vecp, vec)
int        vecp;
char  **vec;
{
    int            reply,
            result,
            sd;
    struct AcSAPstart   acss;
    register struct AcSAPstart *acs = &acss;
    struct AcSAPindication  acis;
    register struct AcSAPindication *aci = &acis;
    register struct AcSAPabort   *aca = &aci -> aci_abort;
    register struct PSAPstart *ps = &acs -> acs_start;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    if (AcInit (vecp, vec, acs, aci) == NOTOK) {
        acs_advise (aca, "initialization fails");
        return NOTOK;
    }
    advise (LLOG_NOTICE, NULLCP,
                "A-ASSOCIATE.INDICATION: <%d, %s, %s, %s, %d>",
                acs -> acs_sd, oid2ode (acs -> acs_context),
                sprintaei (&acs -> acs_callingtitle),
                sprintaei (&acs -> acs_calledtitle), acs -> acs_ninfo);

    sd = acs -> acs_sd;

    for (vec++; *vec; vec++)
        advise (LLOG_EXCEPTIONS, NULLCP, "unknown argument \"%s\"", *vec);
```

```
    reply = startfnx ? (*startfnx) (sd, acs) : ACS_ACCEPT;

    result = AcAssocResponse (sd, reply,
                reply != ACS_ACCEPT ? ACS_USER_NOREASON : ACS_USER_NULL,
                NULLOID, NULLAEI, NULLPA, NULLPC, ps -> ps_defctxresult,
                ps -> ps_prequirements, ps -> ps_srequirements, SERIAL_NONE,
                ps -> ps_settings, &ps -> ps_connect, NULLPEP, 0, aci);

    ACSFREE (acs);

    if (result == NOTOK) {
        acs_advise (aca, "A-ASSOCIATE.RESPONSE");
        return NOTOK;
    }
    if (reply != ACS_ACCEPT)
        return NOTOK;

    if (RoSetService (sd, RoPService, roi) == NOTOK)
        ros_adios (rop, "set RO/PS fails");

    return sd;
}

/*  */

static int  ros_work (fd)
int        fd;
{
    int            result;
    caddr_t out;
    struct AcSAPindication  acis;
    struct RoSAPindication  rois;
    register struct RoSAPindication *roi = &rois;
    register struct RoSAPpreject   *rop = &roi -> roi_preject;

    switch (setjmp (toplevel)) {
        case OK:
            break;

        default:
            if (stopfnx)
                (*stopfnx) (fd, (struct AcSAPfinish *) 0);
        case DONE:
            (void) AcUAbortRequest (fd, NULLPEP, 0, &acis);
            (void) RyLose (fd, roi);
            return NOTOK;
    }

    switch (result = RyWait (fd, NULLIP, &out, OK, roi)) {
        case NOTOK:
            if (rop -> rop_reason == ROS_TIMER)
                break;
        case OK:
        case DONE:
            ros_indication (fd, roi);
            break;

        default:
            adios (NULLCP, "unknown return from RoWaitRequest=%d", result);
    }

    return OK;
```

```
}

/*  */

static int ros_indication (sd, roi)
int        sd;
register struct RoSAPindication *roi;
{
    int            reply,
            result;

    switch (roi -> roi_type) {
        case ROI_INVOKE:
        case ROI_RESULT:
        case ROI_ERROR:
            adios (NULLCP, "unexpected indication type=%d", roi -> roi_type);
            break;

        case ROI_UREJECT:
            {
                register struct RoSAPureject   *rou = &roi -> roi_ureject;

                if (rou -> rou_noid)
                    advise (LLOG_EXCEPTIONS, NULLCP,
                            "RO-REJECT-U.INDICATION/%d: %s",
                            sd, RoErrString (rou -> rou_reason));
                else
                    advise (LLOG_EXCEPTIONS, NULLCP,
                            "RO-REJECT-U.INDICATION/%d: %s (id=%d)",
                            sd, RoErrString (rou -> rou_reason),
                            rou -> rou_id);
            }
            break;

        case ROI_PREJECT:
            {
                register struct RoSAPpreject   *rop = &roi -> roi_preject;

                if (ROS_FATAL (rop -> rop_reason))
                    ros_adios (rop, "RO-REJECT-P.INDICATION");
                ros_advise (rop, "RO-REJECT-P.INDICATION");
            }
            break;

        case ROI_FINISH:
            {
                register struct AcSAPfinish *acf = &roi -> roi_finish;
                struct AcSAPindication  acis;
                register struct AcSAPabort *aca = &acis.aci_abort;

                advise (LLOG_NOTICE, NULLCP, "A-RELEASE.INDICATION/%d: %d",
                        sd, acf -> acf_reason);

                reply = stopfnx ? (*stopfnx) (sd, acf) : ACS_ACCEPT;

                result = AcRelResponse (sd, reply, ACR_NORMAL, NULLPEP, 0,
                            &acis);

                ACFFREE (acf);

                if (result == NOTOK)
                    acs_advise (aca, "A-RELEASE.RESPONSE");
                else
```

```
                    if (reply != ACS_ACCEPT)
                        break;
                longjmp (toplevel, DONE);
            }
        /* NOTREACHED */

        default:
            adios (NULLCP, "unknown indication type=%d", roi -> roi_type);
    }
}

/*  */

static int  ros_lose (td)
struct TSAPdisconnect *td;
{
    if (td -> td_cc > 0)
        adios (NULLCP, "TNetAccept: [%s] %*.*s",
                TErrString (td -> td_reason), td -> td_cc, td -> td_cc,
                td -> td_data);
    else
        adios (NULLCP, "TNetAccept: [%s]", TErrString (td -> td_reason));
}

/*    ERRORS */

void        ros_adios (rop, event)
register struct RoSAPpreject *rop;
char   *event;
{
    ros_advise (rop, event);

    longjmp (toplevel, NOTOK);
}


void        ros_advise (rop, event)
register struct RoSAPpreject *rop;
char   *event;
{
    char    buffer[BUFSIZ];

    if (rop -> rop_cc > 0)
        (void) sprintf (buffer, "[%s] %*.*s", RoErrString (rop -> rop_reason),
                rop -> rop_cc, rop -> rop_cc, rop -> rop_data);
    else
        (void) sprintf (buffer, "[%s]", RoErrString (rop -> rop_reason));

    advise (LLOG_EXCEPTIONS, NULLCP, "%s: %s", event, buffer);
}

/*  */

void        acs_advise (aca, event)
register struct AcSAPabort *aca;
char   *event;
{
    char    buffer[BUFSIZ];

    if (aca -> aca_cc > 0)
        (void) sprintf (buffer, "[%s] %*.*s",
                AcErrString (aca -> aca_reason),
                aca -> aca_cc, aca -> aca_cc, aca -> aca_data);
```

```
    else
        (void) sprintf (buffer, "[%s]", AcErrString (aca -> aca_reason));

    advise (LLOG_EXCEPTIONS, NULLCP, "%s: %s (source %d)", event, buffer,
                aca -> aca_source);
}

/*  */

#ifndef        lint
void        adios (va_alist)
va_dcl
{
    va_list ap;

    va_start (ap);

    _ll_log (pgm_log, LLOG_FATAL, ap);

    va_end (ap);

    _exit (1);
}
#else
/* VARARGS2 */

void        adios (what, fmt)
char    *what,
        *fmt;
{
    adios (what, fmt);
}
#endif


#ifndef        lint
void        advise (va_alist)
va_dcl
{
    int            code;
    va_list ap;

    va_start (ap);

    code = va_arg (ap, int);

    _ll_log (pgm_log, code, ap);

    va_end (ap);
}
#else
/* VARARGS3 */

void        advise (code, what, fmt)
char    *what,
        *fmt;
int         code;
{
    advise (code, what, fmt);
}
#endif
```

```
#ifndef        lint
void        ryr_advise (va_alist)
va_dcl
{
    va_list ap;

    va_start (ap);

    _ll_log (pgm_log, LLOG_NOTICE, ap);

    va_end (ap);
}
#else
/* VARARGS2 */

void        ryr_advise (what, fmt)
char    *what,
        *fmt;
{
    ryr_advise (what, fmt);
}
#endif
```

F.3 Decoders

Decoder modules are constructed using the `flex` template shown in Section F.3.1 for defining scanners and the `bison` template shown in Section F.3.2 for defining parsers.

F.3.1 Scanner

```
%{
/*
 *        scanner definition generated by a2b; don't edit
 *
 *        scanner.skel version 0 (based on cast.scanner.skel version 1.9.3)
 */
#undef YY_DECL
@% the name of the scanner is defined here
static int count = 0, doDummy = 0;
double atof();
long atol();
static resetMode();
%}
%x freeMode
%x fixedMode
%%
        if (doDummy) {
              doDummy = 0;
              return SETFLEXMODE;
        } else if (!(rgcnt[rgflg]) && rgflg) {
              doDummy = 1;
              rgflg--;
              return( EORG );
        } else {
              doDummy = 1;
              switch(lexMode) {
                 case FREE:
                    BEGIN(freeMode);
                    break;
```

```
                case FIXED:
                   BEGIN(fixedMode);
                   count = 0;
                   break;
                default:;
                   return SCANERROR;
                   break;
             }
        }


@% user defined tokens go here

<freeMode>[\n]        { lineNo++; }
<freeMode>[ \t]        { /* do nothing */ }
<freeMode>. { return (SCANERROR); }
<fixedMode>\n        { /* do nothing */ }

<fixedMode>.        {
        /* BUG: detecting missing fixed values;
         *      only missing fixed values consisting of
         *        2 or more blanks will be detected.
         *        Should be dealt with better.
         */
        char *tmpstr = (char *)malloc(sizeOfNext+1);
        int missFxdVal = *yytext == ' ';
        tmpstr[count] = *yytext; count++;
        while ((count < sizeOfNext) && (tmpstr[count-1] != EOF)) {
                tmpstr[count++] = input();
                missFxdVal = missFxdVal && (tmpstr[count-1] == ' ');
        }
        if (missFxdVal && (sizeOfNext != 1))
              return(MISSFXDVAL);
        else if (count == sizeOfNext) {
              tmpstr[count] = '\0';
@% flex debugging statement : fprintf(stderr, "-- accepting \"%s\" in fixedMode\n", tmpstr);
              switch(rtnType) {
                 case IA5STRING:
                    (*yylvalp).string = ( tmpstr ? str2qb(tmpstr,strlen(tmpstr)+1,0) : NULL);
                    resetMode;
                    return( IA5STRING );
                    break;
                 case INT:
                    (*yylvalp).integer = atol(tmpstr);
                    resetMode;
                    return( INT );
                    break;
                 case REAL:
                    (*yylvalp).real = atof(tmpstr);
                    resetMode;
                    return( REAL );
                    break;
              }
        }
        else /* EOF before reading fixed field size -- terminate */
              yyterminate();
}
@% Any user defined kerning tokens go here
%%
static int resetMode()
{
      lexMode = -1;
      rtnType = 0;
```

```
        sizeOfNext = -1;
        BEGIN(INITIAL);
}
```

F.3.2 Parser

```
%{
/*
 *        parser definition generated by a2b; don't edit
 *
 *        parser.skel version 0.1 (based on cast.parser.skel 1.9)
 */
#include <stdio.h>

#define FREE 0
#define FIXED 1
#define UNDEF 2

@% put here is:..-types.h; scanner decl; ..Data var; and initial scan mode
static yyerror();
static int yylex();
#define yyparse static parse

#define maxrgflg 20
static int rgcnt[maxrgflg], rgflg = 0;
#define INITRG(A) rgcnt[++rgflg] = A;
#define DECRG rgcnt[rgflg]--;
int casCounter[50];

%}
%pure_parser
%union {
        long integer;
        double real;
        struct qbuf *string;
/*
 *        user defined type declarations follow
 */
@% the union declarations go here
}
@% the %type decls go here
%token <string> IA5STRING ALPHANUM
%token <integer> INT
%token <real> REAL
%token SETFLEXMODE SCANERROR EORG MISSFXDVAL

%start CASstart

/*
 *         user defined tokens go next
 */
@% user defined tokens go here
%%
@% grammar rules go here
%%
static FILE *yyin = (FILE *) 0;
static int yy_init = 1, yy_start = 0;
static char *yytext;

@% parser declaration goes here
{
        int result = 0;
```

```
        yyin = fopen(fn, "r");
        yy_init = 1;
        yy_start = 0;
@% set yydebug=1 if debugging the parser or the decoder. Set any counters

        result = (yyin ? OK : NOTOK);

        if (result == OK) result = parse();

@% Data variable assignment occurs here

        fclose(yyin); yyin = NULL;

        return result;
}
static yyerror(s) char *s;
{
        fprintf(stderr,"%s: near line %d last token read was %s\n",s, lineNo, yytext);
}
```

F.4 Translators

F.4.1 Miranda translator template

```
#! /pcks/miranda/mira -exp
doTranslation

@% include files declaring data types

@% decodeFile :: [char] ->
decodeFile fileName
       = readvals fileName

encodeFile fileName dataString
      = [ Tofile fileName dataString ]

@% the translator

doTranslation
    = encodeFile outFile (show data)
      where
                outFile = $*!2
          inFile = $*!1
@%          data = >translator< (decodeFile inFile)
```

Acknowledgements

I would like to thank the many people who have assisted me while I completed this research. First, Prof. John Penny for his supervision. Second, the staff of the Computer Science Department at Canterbury University. In particular: Dr. Neville Churcher who supervised my research when Prof. John Penny was away on study leave and discussed many aspects of my research associated with data modelling and database management systems; and Dr. Bruce McKenzie, who advised me on the mysteries of parser construction and grammars.

I would also like to thank Dr. Greg Ewing for the time and assistance he has given me to improve the software tool `a2b` and for constructing the tool `kerngen`; and David Gee for reading an early draft of the thesis.

Finally, I would like to express my deepest appreciation for the support given to me by my family and friends. They eased a very difficult time in my life as I struggled with the trials and tribulations of being a university student.

References

Abbott, R. J. (1989), 'Set notation as a language to specify data transformation programs', *Software–Practice and Experience* **19**(6), 593–606.

Adobe (1985), *POSTSCRIPT Language Reference Manual*, Addison-Wesley Publishing Company Inc., Reading, Massachusetts.

Barron, D. W. & Rees, M. (1987), *Text processing and typesetting with Unix*, International Computer Science Series, Addison-Wesley Publishing Company Inc., Reading, Massachusetts.

Batini, C., Lenzerini, M. & Navathe, S. B. (1986), 'A comparative analysis of methodologies for database schema integration', *ACM Computing Surveys* **18**(4), 323–364.

Cadre (1987), Teamwork/SA user's manual, Technical Report, Cadre Technology Inc., Providence, Rhode Island.

Chen, P. P. (1976), 'The Entity-Relationship Model — towards a unified view of data', *ACM Transactions on Database Systems* **1**(1), 9–36.

Codd, E. F. (1970), 'A Relational model for large shared data banks', *Communications of the ACM* **13**(6), 377–387.

Corbett, R. (1989), *Bison Reference Manual*, Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

CSIRONET (1986), *Command Driven Colourmap, User's Guide*, 1st edn, CSIRONET Graphics System Section, Canberra.

Date, C. J. (1990), *An Introduction to Database Systems*, Vol. 1, 5th edn, Addison-Wesley Publishing Company Inc., Reading, Massachusetts.

Egenhofer, M. J. & Herring, J. R. (1991), High-level spatial data structures for GIS, *in* D. J. Maguire, M. F. Goodchild & D. W. Rhind, eds, 'Geographical Information Systems: principles and applications', Vol. 1, Longman Scientific & Technical, London.

Egenhofer, M. J., Frank, A. U. & Jackson, J. (1989), A topological data model for spatial databases, *in* 'Lecture Notes in Computer Science 409: Proceedings of the 1st Symposium, SSD', Springer-Verlag, Berlin, pp. 271–286.

Elmasri, R. & Navathe, S. B. (1989), *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company Inc., 2727 Sand Hill Road, Menlo Park, California 94025.

Evangalatos, T., Jiwani, Z., McKellar, D. & O'Brien, C. D. (1989), The telecommunication of map and chart data, *in* 'Auto Carto 9 Ninth International Symposium on Computer-Assisted Cartography', American Society for Photogrammetry and Remote Sensing, American Congress on Surveying and Mapping, pp. 754–763.

Evangalatos, T. V. & Allam, M. M. (1991), Canadian efforts to develop spatial data exchange standards, *in* H. Moellering, ed., 'Spatial Database Transfer Standards: Current International Status', Elsevier Applied Science, London and New York, pp. 45–67.

Fegeas, R. G., Cascio, J. L. & Lazar, R. A. (1992), 'An overview of FIPS 173, The Spatial Data Transfer Standard', *Cartography and Geographic Information Systems* ***19*** *(5).*

Fischer, C. N. & LeBlanc Jr., R. J. (1988), *Crafting A Compiler*, The Benjamin/Cummings Publishing Company Inc., 2727 Sand Hill Road, Menlo Park, California 94025.

Fosnight, E. A. & van Roessel, J. W. (1985), Vector data interfacing at the EROS data center; RIM to ARC/INFO and related interfaces, Technical Report, EROS Data Center, Sioux Falls, South Dakota 57198.

Frank, A. U. (1992), 'Spatial concepts, geometric data models, and geometric data structures', *Computers and Geosciences* **18**(4), 409–417.

Gawkowski, J. A. & Mamrak, S. A. (1992), Towards a universal framework for data translation, Technical Report, Dept. Computer and Info. Sci., Ohio State University.

Geological Survey, U. S. (1990), Digital Line Graphs from 1:2,000,000-Scale Maps, *in* 'Data Users Guide', Vol. 3 of *National Mapping Program Technical Instructions*, Department of Interior, U. S. Geological Survey.

Geological Survey, U. S. (1992), *Spatial Data Transfer Standard (SDTS) (FIPS 173)*, Department of Interior, U. S. Geological Survey, 561 National Center, Reston, Virginia 22092, USA.

GeoVision (1986), *Data Translation Guide*, GeoVision, Ottawa.

Goodchild, M. F. (1992), 'Geographical modeling', *Computers and Geosciences* **18**(4), 401–408.

Holroyd, F. & Bell, S. B. M. (1992), 'Raster GIS: models of raster encoding', *International Journal of Geographical Information Systems* **18**(4), 419–426.

Hopcroft, J. E. & Ullman, J. D. (1979), *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Series in Computer Science, Addison-Wesley Publishing Company Inc., Reading, Massachusetts.

Ingres (1989), *INGRES/NET User's and Administrator's Guide, Release 6.3*, Ingres Corporation, 1080 Marina Village Parkway Alameda, California 94501.

Ingres (1993), 'INGRES announces new database capabilities', *Ingres World.*

ISO2022 (1986), Information processing – ISO 7-bit and 8-bit coded character sets – Code extension techniques (ISO 2022), *in* 'Information processing systems – Open Systems Interconnection', International Organisation for Standardization.

ISO2375 (1985), Data processing – Procedure for registration of escape sequences (ISO 2375), *in* 'Information processing systems — Open Systems Interconnection', International Organisation for Standardization.

ISO7498 (1984), Basic reference model (incorporating connectionless mode transmission) (ISO 7498), *in* 'Information processing systems — Open Systems Interconnection', International Organisation for Standardization.

ISO8211 (1985), Specification for a data descriptive file for information interchange (ISO 8211), *in* 'Information processing systems — Open Systems Interconnection', International Organisation for Standardization.

ISO8613-1 (1989), Office Document Architecture (ODA) and interchange format – Part 1:Introduction and general principles, *in* 'Information processing systems — Text and Office Systems', International Organisation for Standardization.

ISO8822 (1988), Connection-oriented presentation service definition, *in* 'Information processing systems — Open Systems Interconnection', International Organisation for Standardization.

ISO8824 (1987), Specification for Abstract Syntax Notation One (ASN.1) (ISO 8824), *in* 'Information processing systems — Open Systems Interconnection', International Organisation for Standardization.

ISO8825 (1987), Specification for basic encoding rules for Abstract Syntax Notation One (ASN.1) (ISO 8825), *in* 'Information processing systems — Open Systems Interconnection', International Organisation for Standardization.

ISO9072-1 (1988), Remote Operations Part 1: Model, Notation, and Service Definition. (ISO 9072-1), *in* 'Information processing systems — Text Communication', International Organisation for Standardization.

ISO9072-2 (1988), Remote Operations Part 2: Protocol Specification. (ISO 9072-2), *in* 'Information processing systems — Text Communication', International Organisation for Standardization.

Johnson, S. C. (1979), *Unix Time-Sharing System: Unix Programmer's Manual: Yacc: yet another compiler-compiler*, 7th edn, ATT Bell Laboratories, Murray Hill, New Jersey.

Kemp, Z. (1990), An object-oriented data model for spatial data, *in* K. Brassel & H. Kishimoto, eds, 'Proceedings of the 4th International Symposium on Spatial Data Handling', pp. 659–668.

Kernighan, B. W. & Ritchie, D. M. (1978), *The C Programming Language*, Prentice-Hall Software Series, Prentice-Hall, Englewood Cliffs, New Jersey 07632.

Lamport, L. (1986), *LaTeX: A Document Preparation System*, Addison-Wesley Publishing Company Inc., Reading, Massachusetts.

Lesk, M. E. & Schmidt, E. (1979), *Unix Time-Sharing System: Unix Programmer's Manual: Lex — A lexical analyzer generator*, 7th edn, ATT Bell Laboratories, Murray Hill, New Jersey.

Maffini, G. (1987), 'Raster versus vector data encoding and handling: a commentary', *Photgrammetric engineering and remote sensing* **53**(10), 1397–1398.

Mamrak, S. A., Kaelbling, N. J., Nicholas, C. K. & Share, M. (1989), 'Chameleon: a system for solving the data translation problem', *IEEE Transactions on Software Engineering* **15**(9), 1090–1108.

Mark, D. M. (1978), Concepts of data structure for digital terrain models, *in* 'Proceedings of the Digital Terrain Models (DTM) Symposium', American Society of Photogrammetry, Falls Church, Va., pp. 24–31.

Martin, J. & McClure, C. (1985), *Diagramming techniques for analysts and programmers*, Prentice-Hall, Englewood Cliffs, New Jersey 07632.

McKellar, D. G., O'Brien, C. D. & Lalonde, W. T. (1990), An Architecture for the Exchange of Geographic Data, *in* 'GIS for the 1990's Proceedings', Canadian Institute of Surveying and Mapping, pp. 893–900.

Milne, P., Milton, S. & Smith, J. L. (to appear), 'Geographical object-oriented databases — a case study', *International Journal of Geographical Information Systems.*

Naur, P., Backus, J. W., Katz, C., Rutishauser, H., Wegstein, J. H., Bauer, F. L., McCarthy, J., Samelson, K., van Wijngaarden, A., Green, J., Perlis, A. J., Vauquois, B. & Woodger, M. (1963), 'Revised report on the algorithmic language ALGOL 60', *Communications of the ACM* **6**, 1–17.

Nyerges, T. (1984), Digital cartographic data standards: alternatives in data organization, Progress report of the workgroup on data organization of the national committee for digital cartographic data standards, National Committee for Digital Cartographic Data Standards.

Nyerges, T. (1989), 'Schema integration analysis for GIS', *International Journal of Geographical Information Systems* **3**(2), 153–183.

Pascoe, R. T. (1989), Data translation between geographic information systems, M.Sc. thesis, Department of Computer Science, University of Canterbury.

Pascoe, R. T. & Churcher, N. (1990), 'Modelling the sharing of geographic data', *NZ Journal of Computing* **2**(1), 45–53. Reprinted from the Proceedings of the 2nd Annual Colloquium of the Spatial Information Research Centre, University of Otago, Dunedin, New Zealand.

Pascoe, R. T. & Penny, J. P. (1990), 'Construction of interfaces for the exchange of geographic data', *International Journal of Geographical Information Systems* **4**(2), 147–156.

Pascoe, R. T. & Penny, J. P. (1993), Transforming geographic data between different concrete representations, *in* G. Gupta, G. Mohay & R. Topor, eds, 'Proceedings of the Sixteenth Australian Computer Science Conference', pp. 653–633.

Pascoe, R. T. & Penny, J. P. (to appear), 'Constructing interfaces between (and within) geographical information systems', *International Journal of Geographical Information Systems*. Accepted for publication in April 1994.

Paxson, V. (1990), *Flex–fast lexical analyzer generator*, Computer Systems Engineering, Bldg. 46A, Room 1123, University of California, Berkeley, CA 94720. Documentation provided with source code for Flex.

Penny, J. P. (1986), 'Relational methods for format conversion of map data', *Cartography* **15**(1), 26–34.

Peucker, T. K., Fowler, R. J., Little, J. J. & Mark, D. M. (1978), The triangulated irregular network, *in* 'Digital Terrain Models (DTM) symposium', American society of photgrammetry, pp. 516–540.

Peuquet, D. (1984), 'A conceptual framework and comparison of spatial data models', *Cartographica* **21**, 66–113.

Raper, J. F. & Kelk, B. (1991), Three-dimensional gis, *in* D. J. Maguire, M. F. Goodchild & D. W. Rhind, eds, 'Geographical Information Systems: principles and applications', Vol. 1, Longman Scientific & Technical, London.

Razouk, R. R. (1987), A guided tour of P-NUT, Technical Report #86-25, Dept. Information and Computer and Science, University of California, Irvine.

Rose, M. T., Onions, J. P. & Robbins, C. J. (1991), *The ISO Development Environment: Users Manual*, 7th edn, Performance Systems International and X-Tel Services, X-Tel Services Ltd. University Park, Nottingham, NG7 2RD, UK.

Samet, H. (1984), 'The quadtree and related hierarchical data structures', *ACM Computing Surveys* **16**, 187–260.

Samet, H. (1990), *The Design and Analysis of Spatial Data Structures*, Addison-Wesley Publishing Company Inc., Reading, Massachusetts.

Shapiro, M., Westervelt, J., Gerdes, D., Larsen, M. & Brownfield, K. R. (1992), *Grass 4.0 Programmer's Manual (Draft)*, U.S. Army Construction Engineering Research Laboratory, GRASS Information Center, USACERL, P.O. Box 9005, Champaign, IL, 61826-9005.

Shu, N. C., Housel, B. C., Taylor, R. E., Ghosh, S. P. & Lum, V. Y. (1977), 'EXPRESS: A Data EXtraction, Processing, and REStructuring System', *ACM Transactions on Database Systems* **2**, 134–174.

Smith, B. & Wellington, J. (1992), *Initial Graphics Exchange Specification (IGES), Version 3.0*, U. S. Department of Commerce.

Smith, J. M. & Smith, D. C. (1977), 'Database abstractions: aggregation and generalization', *ACM Transactions on Database Systems* **2**(2), 105–133.

Stallman, R. M. & Pesch, R. H. (1989), *Using GDB: A Guide to the GNU Source-Level Debugger*, Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Star, J. L. & Dickinson, H. (1990), INTRODUCTION to GIS, *in* M. F. Goodchild & K. K. Kemp, eds, 'NCGIA Core Curriculum', National Center for Geographic Information and Analysis.

Stevens, R. J., Lehar, A. F. & Preston, F. H. (1983), 'Manipulation and presentation of multidimensional image data using the Peano scan', *IEEE Trans. on Pattern Analysis and Machine Intelligence* **5**, 520–526.

StoneBraker, M. (1992), *Postgres Reference Manual, Version 4.0*, University of California, Berkeley. This manual is distributed with the Postgres source code.

Sudkamp, T. A. (1988), *Languages and Machines*, Addison-Wesley Publishing Company Inc., Reading, Massachusetts.

Sun (1988), *Debugging Tools*, Sun Microsystems, Part Number: 800-1775-10, Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043.

Taylor, R. W. (1982), 'Experiences using generalised data translation techniques for database interchange', *Computers and Standards* **1**, 111–118.

Tichy, W. F. (1985), 'RCS—A system for version control', *Software-Practice & Experience* **15**(7), 637–654.

Tsichritzis, D. C. & Klugs, A. (1978), 'The ANSI/X3/SPARC DBMS Framework: Report of the Study Group on Data Base Management Systems', *Information Systems.*

Tsichritzis, T. C. & Lochovsky, F. H. (1977), *Data Base Management Systems*, Academic Press, New York.

Turner, D. (1986), 'An Overview of Miranda', *ACM SIGPLAN Notices* **21**(12), 158–166.

Ullman, J. D. (1988), *Principles of Database and Knowledge-base Systems*, Vol. 1 of *Principles of Computer Science*, Computer Science Press, Inc, 1803 Research Boulevard, Rockville, Maryland 20805, USA.

Unilogic, L. (1984), *Scribe Document Production System User Manual.*

van Roessel, J., Bankers, D., Connochioli, V., Doescher, S., Fosnight, G., Wehde, M. & Tyler, D. (1986), vector data structure conversion at the EROS data center, final report, phase I, Technical Report, EROS Data Center, Sioux Falls, South Dakota 57198.

van Roessel, J. W. (1987), 'Design of a spatial data structure using the relational normal forms', *International Journal of Geographical Information Systems* **1**(1), 33–50.

van Roessel, J. W. & Fosnight, E. A. (1985), A relational approach to vector data structure conversion, *in* 'Proceedings Auto-carto Conference', Washington.

Waugh, T. C. & Healy, R. G. (1986), The GEOLINK system, interfacing large systems, *in* M. Blakemore, ed., 'Auto Carto London', pp. 76–85.

Waugh, T. C. & McCalden, J. (1983), *GIMMS Reference Manual*, 4.5 edn, GIMMS Ltd, 30 Keir Street, Edinburgh EH3 9EU, Scotland.

Worboys, M. F. (1992), 'A generic model for planar geographical objects', *International Journal of Geographical Information Systems* **6**(5), 353–372.

Worboys, M. F., Hearnshaw, H. M. & Maguire, D. J. (1990), 'Object-oriented data modelling for spatial databases', *International Journal of Geographical Information Systems* 4(4), 369–383.